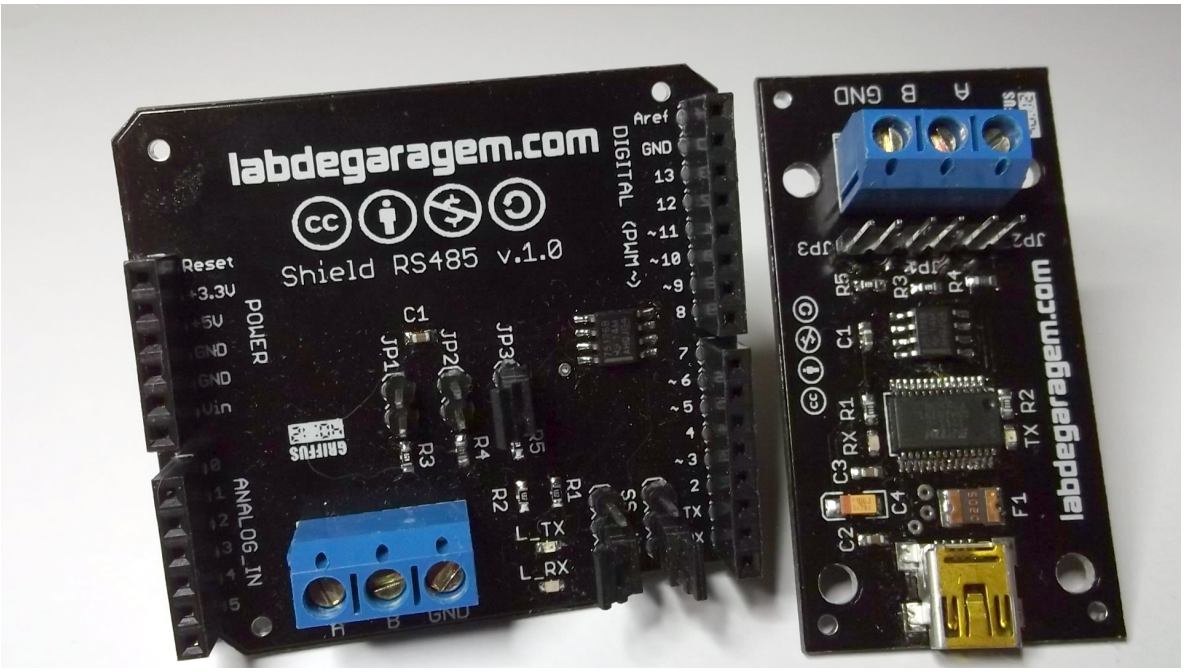
Rua Berta, 60 – Vila Mariana
São Paulo – SP – Brasil
(11) 3804-0126
Rede Social: http://www.labdegaragem.com.br
Loja Virtual: http://loja.labdegaragem.com.br

KIT Shield RS485 + Conversor USB/RS485



Manual do Usuário

Obrigado por adquirir o KIT Shield RS485 + Conversor USB/RS485 do Laboratório de Garagem.

Através deste manual, você conhecerá maiores detalhes sobre este produto, para iniciar seus projetos de integração do Arduino com redes RS485.

Conteúdo deste documento

- Introdução
- Descrição do KIT
- Características técnicas

- Instalação do Shield no Arduino
- Instalação do conversor USB/RS485 ao PC
- Teste Inicial
- Usando o Kit com o SCADABR
- Apêndice I - Introdução ao barramento RS485
- Apêndice II - Introdução ao protocolo MODBUS padrão RTU
- Apêndice *III* - Sketch para teste de comunicação serial RS485
- Apêndice IV - Sketch para teste com o SCADABR e Arduino escravo MODBUS
- Referências

Introdução

De um garagista para outro...

Este Kit foi desenvolvido para permitir a conectividade do Arduino às redes RS485, permitindo aplicações em sistemas de automação onde seja necessária a conexão a longas distâncias do Arduino com um PC que esteja rodando um programa de supervisão, com outros Arduinos ou qualquer equipamento em uma rede RS485.

O kit é uma solução prática para conectar microcontroladores Arduino a um computador e pode ser utilizado pra fins de automação residencial, industrial, agronegócios ou didáticos. Leia com atenção este manual e se surgirem dúvidas não deixe de entrar em contato, pois faremos o possível para ajudar.

Descrição do KIT

O KIT é composto por um shield para Arduino e um conversor USB/RS485 conforme vemos na figura 1.

Rua Berta, 60 – Vila Mariana
São Paulo – SP – Brasil
(11) 3804-0126
Rede Social: http://www.labdegaragem.com.br
Loja Virtual: http://loja.labdegaragem.com.br

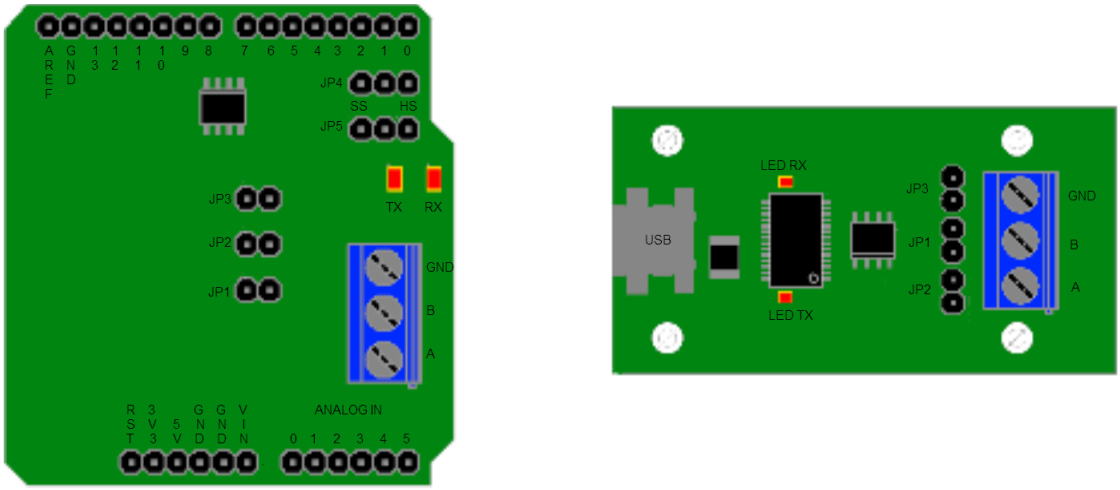


Fig. 1: Vista das placas do KIT.

O shield contém um chip transceiver RS485 que faz o trabalho de conversão dos dados de TTL para RS485 e vice versa, LEDs para indicar o fluxo do tráfego na rede, bornes para ligar os cabos do barramento da rede, jumpers para selecionar uso de resistores de terminação e bias da rede e jumpers para escolher entre usar uma porta de comunicação serial física do ATmega do Arduino ou usar uma porta soft serial. Para a comunicação pela hard serial (HS) são utilizados os pinos digitais 0 e 1 do Arduino. Para a comunicação usando soft serial (SS) são usados os pinos digitais 2 e 3 do Arduino. O pino digital 6 é usado para comutação entre os modos de transmissão e recepção do transceiver do shield.

O conversor USB/RS485 usa o chip FT232RL para criar um dispositivo serial em qualquer microcomputador (porta COM) que pode ser acessada pelo usuário ou programas. O conversor também possui o mesmo circuito do transceiver RS485 com os jumpers, resistores, LEDs e bornes presente no shield.

Função dos Jumpers do Shield e do conversor:

- JP1 - Liga ou desliga o resistor de pull up do barramento;
- JP2 - Liga ou desliga o resistor de terminação do barramento;
- JP3 - Liga ou desliga o resistor de pull down do barramento;
- JP4 - Seleciona o RX da HS ou da SS;

- JP5 - Seleciona o TX da HS ou da SS.

Observações:

- JP1 e JP3 só podem ser ligados em um único dispositivo da rede e devem ser selecionados igualmente, os dois abertos ou os dois fechados.
- JP2 só pode estar ligado se o dispositivo estiver em uma das duas extremidades do barramento, devendo obrigatoriamente ficar desligado quando estiver no meio da rede.

- JP4 e JP5 devem ser selecionados igualmente, ou para HS ou para SS.

- JP4 e JP5 não podem estar ligados em HS durante o upload de um sketch para o Arduino.

- Os Pinos Digitais 0 e 1, ou 2 e 3 e o pino 6 não poderão ser utilizados por outro periférico.

- O upload de um sketch sempre deve ser feito pelo canal USB nativo do Arduino, não sendo possível fazer-lo pela rede RS485.

Características técnicas

- O Kit atende a norma Standard EIA RS-485 Physical media.
- Opera com qualquer protocolo de comunicação serial com suporte a RS485 como o protocolo MODBUS RTU entre outros.

- Taxa de transmissão suportada de 4800bps a 115200bps.

- Topologia de rede ponto a ponto, ou multi-ponto com cabo de par trançado.

- Modo de operação balanceado diferencial half duplex.

- Até 32 dispositivos são suportados na mesma rede.

- Comprimento máximo do cabo da rede de 1200m (4000 pés).

- Níveis de tensão no barramento RS485 de -7 V a +12 V. Mark (1) tensões positivas (B-A > +200 mV) Space (0) tensões negativas (B-A < -200 mV).

- O chip SN75176 é usado para a conversão TTL/RS485.

- **Utiliza os pinos digitais 0 e 1, ou 2 e 3 do Arduino para comunicação serial com o shield e o pino digital 6 para comutação entre os modos de transmissão e recepção. Obs. Os Pinos Digitais 0 e 1, ou 2 e 3 e o pino 6 não poderão ser utilizados por outro periférico**.

- A alimentação do shield é feita através dos pinos 5 V e GND da placa do Arduino.

- Disponibiliza os pinos do Arduino em dois barramentos laterais, facilitando aplicações que possam utilizar os pinos de I/O do módulo.

- O conversor USB/RS485 utiliza o chip FTD232RL para a conversão de USB para TTL e é alimentado pela porta USB do computador.

Instalação do Shield no Arduino

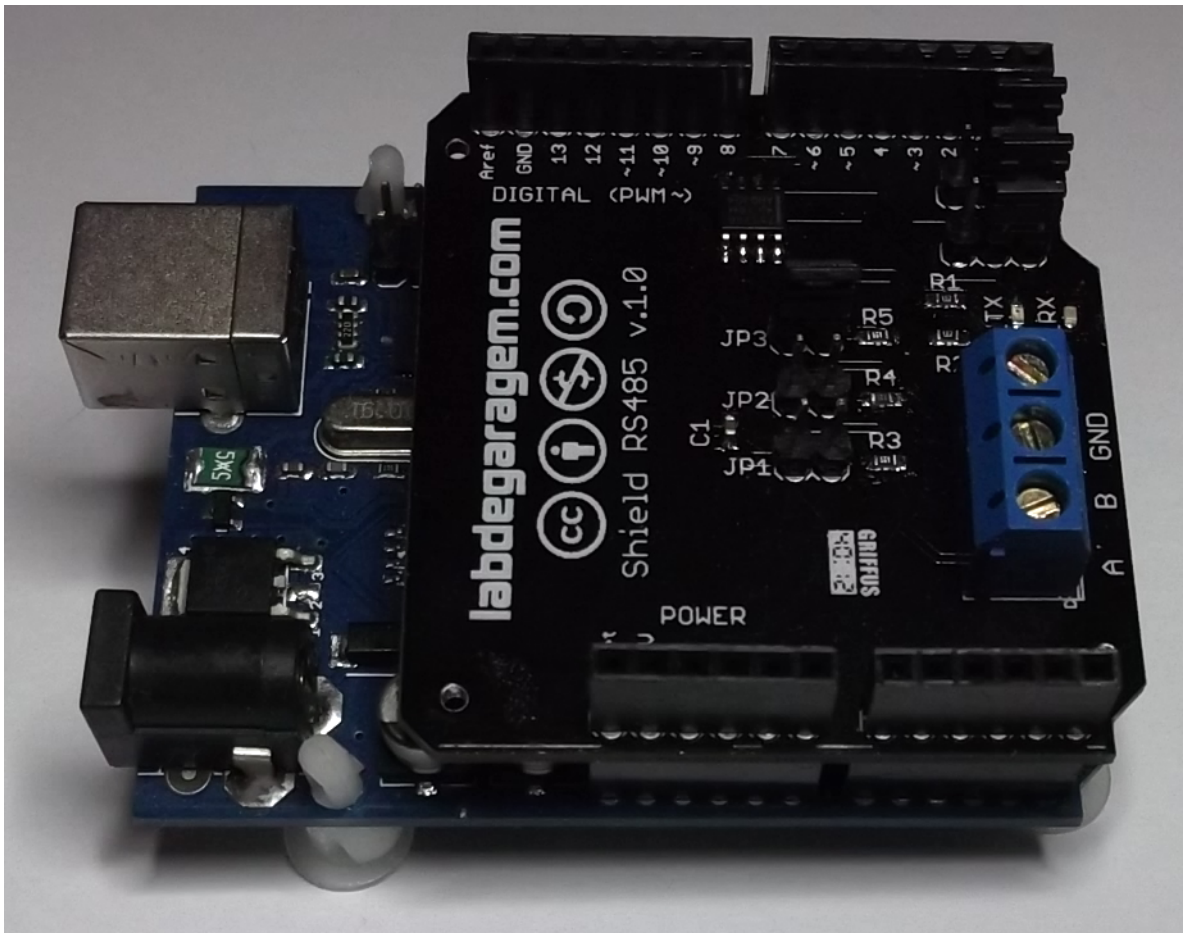


Fig. 2: Detalhe dos jumpers e conectores do Shield RS485 encaixado sobre o Arduino.

1. Encaixar o shield conversor RS485/TTL ao Arduino como na figura 2;
2. Ligar o cabo positivo ou A do barramento da rede RS485 ao borne A do conversor RS485/TTL usando uma chave de fenda;
3. Ligar o cabo negativo ou B do barramento da rede RS485 ao borne B do conversor RS485/TTL usando uma chave de fenda;
4. Ligar o borne GND à malha do cabo do barramento;
5. Abra os jumpers JP4 e JP5;
6. Selecione o sketch desejado na IDE do Arduino e faça o upload para o Arduino usando o cabo USB do Arduino, sempre verifique que JP4 e JP5 estão abertos antes de fazer o upload;
7. Religue os jumpers JP4 e JP5 selecionando para HS ou SS, conforme esteja no sketch usado no Arduino;
8. Ligar os jumpers JP1, JP2 e JP3 conforme a necessidade, ver as observações na seção Descrição do KIT para mais detalhes.

Instalação do conversor USB/RS485 no PC

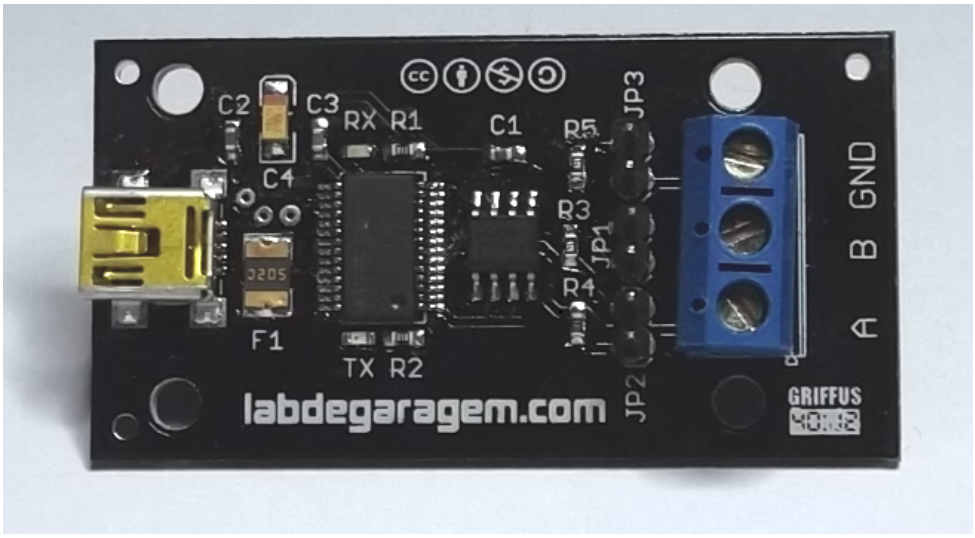


Fig. 3: Detalhe dos jumpers e conectores do Conversor USB/RS485.

1. Ligar o cabo positivo ou A do barramento da rede RS485 ao borne A do conversor USB/ RS485 usando uma chave de fenda;

2. Ligar o cabo negativo ou B do barramento da rede RS485 ao borne B do conversor USB/ RS485;

3. Ligar o borne GND à malha do cabo do barramento;

4. Ligar o cabo USB ao conector USB do conversor e a uma porta USB livre do PC;

5. Aguarde o PC reconhecer o novo dispositivo e quando abrir a tela pedindo a indicação do caminho para o drive de instalação, aponte para a pasta do PC onde está instalado a IDE do Arduino e procure pela pasta FTDI USB Drivers dentro da pasta DRIVERS, normalmente será algo parecido com:

   X:\arduinoYYY\drivers\FTDI USB Drivers;

6. Após a instalação ser concluída, verifique qual porta foi criada usando o gerenciador de dispositivos do Windows.

Configuração

- O usuário deve configurar igualmente todos os parâmetros das portas de comunicação de todos os dispositivos ligados a rede RS485, observando os valores de bits por segundo (baud rate), número de bits dos dados transmitidos, paridade e stop bits.

- Todos os dispositivos devem estar usando o mesmo protocolo de comunicação.

- O pino digital (6) do Arduino terá função de enable (EN) que habilita o conversor RS485/TTL para o modo de transmissão quando estiver em nível lógico 1 (5V) ou para modo de recepção quando o pino de enable estiver em nível lógico 0 (0V).

- Os LEDs TX e RX do shield e do conversor indicam o tráfego da rede.

Rua Berta, 60 – Vila Mariana
São Paulo – SP – Brasil
(11) 3804-0126
Rede Social: http://www.labdegaragem.com.br
Loja Virtual: http://loja.labdegaragem.com.br

Teste Inicial

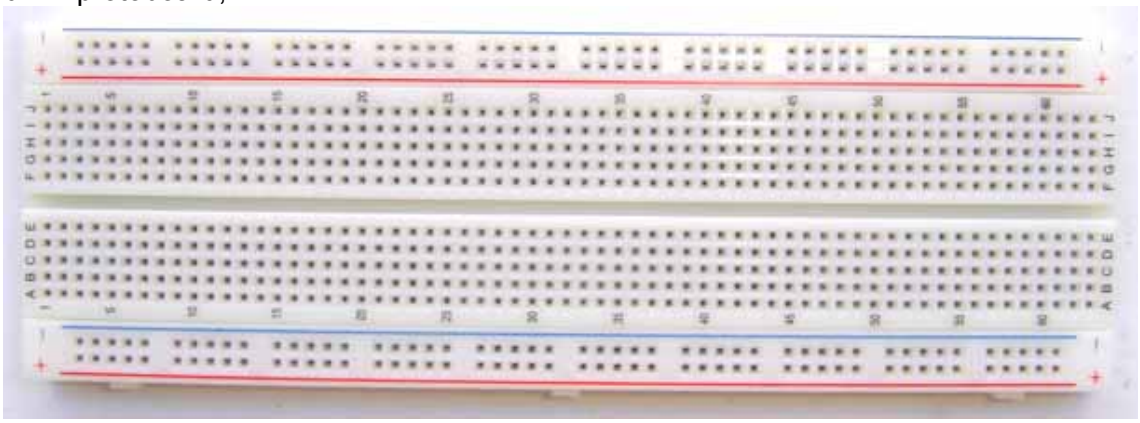
Para testar o funcionamento correto dos módulos e a comunicação entre o PC e o Arduino a forma mais simples é usando o SERIAL MONITOR que está disponível na IDE do Arduino e montando a aplicação a seguir.
Para esta montagem você vai precisar de alguns componentes eletrônicos:

a. 1 potenciômetro rotativo de 16mm de 10kΩ;

b. 1 resistor de 10kΩ x 1/8W;

c. 1 chave táctil (ou um push botton de qualquer tipo);

d. 1 protoboard;



e. Jumpers para ligação;

f. Cabo manga para o barramento (na falta deste usar cabo CCI de 2 pares);

Rua Berta, 60 – Vila Mariana
São Paulo – SP – Brasil
(11) 3804-0126
Rede Social: http://www.labdegaragem.com.br
Loja Virtual: http://loja.labdegaragem.com.br

g. Você também vai precisar de algumas ferramentas como uma chave de fenda de 1/8 x 4” e um alicate de bico fino.

De posse das ferramentas e dos componentes, siga os passos abaixo:

1. Instalar o shield ao Arduino e o conversor ao PC conforme mostrado nos itens instalação e configuração;
2. Montar o circuito nos conectores do shield usando os componentes conforme mostrado na figura 4.
3. Ligar o potenciômetro de 10k com o cursor ligado a entrada analógica 0 e os pinos externos ligados ao GND e 5V;
4. Ligar a chave táctil entre o 5V e a entrada digital 4 com resistor de pull down de 10k;
   ATENÇÃO: Revisar todas as ligações com muito cuidado antes de energizar o circuito, para evitar possíveis danos aos aparelhos!!!
5. Ligar uma fonte de alimentação de 7 a 12 VCC ao conector de alimentação externa do Arduino.
6. Abrir um novo sketch na IDE do Arduino e copiar o texto do apêndice III

7. Faça o upload para o Arduino conectando o cabo USB no conector do Arduino e a uma porta USB do PC. Não se esqueça de desligar os jumpers JP4 e JP5 durante o upload do sketch religando-os em seguida;

8. Assim que a mensagem confirmando que o upload terminou corretamente, desligue o cabo USB do Arduino e feche a IDE do Arduino;

9. Ligue o conversor USB/RS485 ao PC e reabra a IDE do Arduino, abra o sketch do teste e na aba Tools / Serial Port selecione a porta COM do conversor USB/RS485.

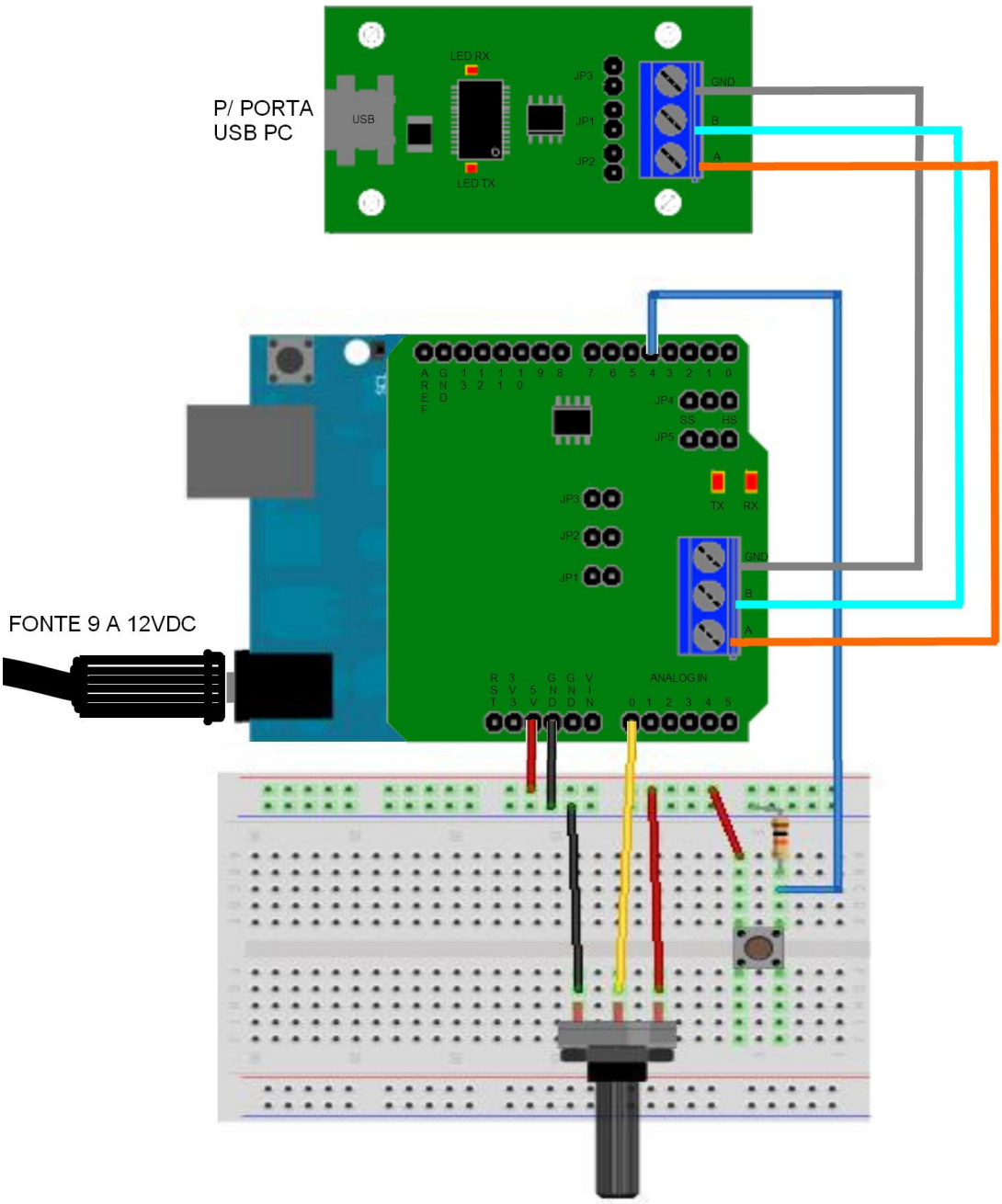


Fig. 4: Montagem para o teste de comunicação RS485 entre o Arduino e o PC.

10. Neste sketch o Arduino ficará aguardando o recebimento de um caractere digitado no teclado do PC pelo usuário que deve estar usando um programa de terminal como por exemplo o Serial Monitor da IDE do Arduino. Assim que for recebido um caractere, o Arduino irá enviar de volta ao PC o valor presente na entrada analógica A0 (variando de 0 a 255), o estado da entrada digital D4 (0 ou 1) e o caractere que foi digitado;
11. Abra o SERIAL MONITOR na IDE do Arduino, e ajuste a velocidade para 9600 baud. Agora digite a tecla “a” e pressione a tecla Send (ou tecle ENTER), se tudo estiver correto, você terá uma tela parecida com a da figura 5. Faça o teste com outras teclas, mas digite e envie apenas uma de cada vez e aguarde o retorno do Arduino, não se esqueça que o RS485 é half duplex, e o seu dedo é o protocolo de comunicação.
12. Teste a comunicação com outras velocidades mudando o baud rate no sketch na linha Serial.begin(9600); faça o novo upload repetindo os passos de 7 a 11 e mudando também a velocidade no Serial Monitor.

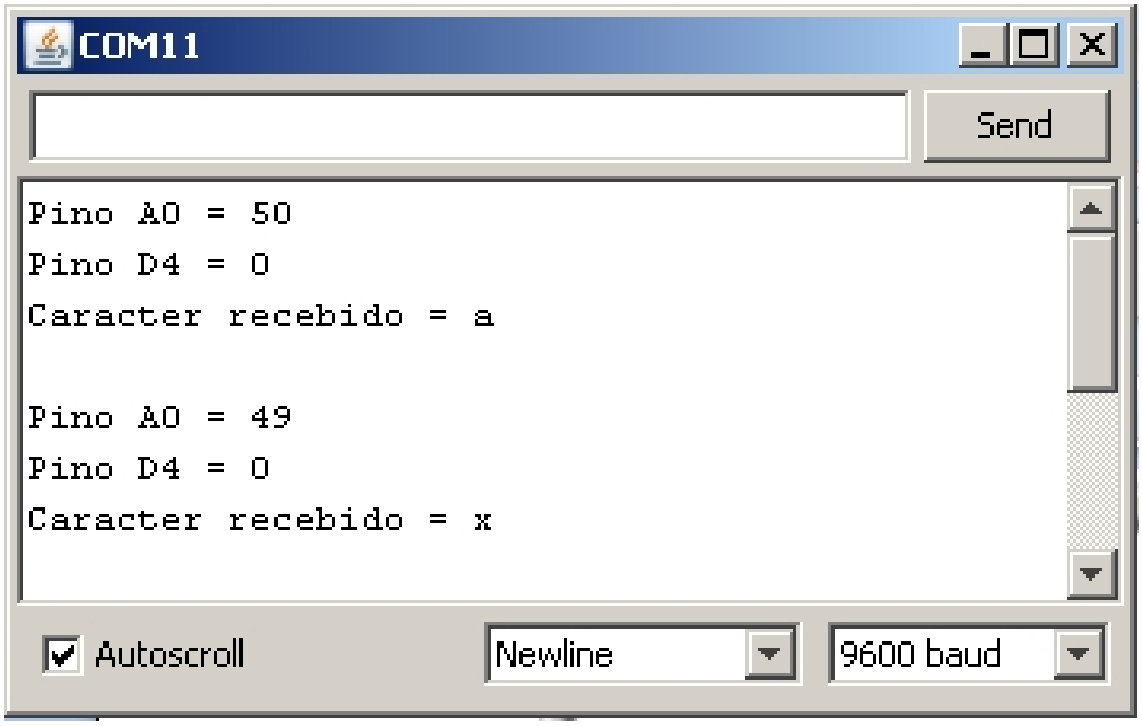


Fig. 5: Teste de comunicação RS485 entre o Arduino e o PC usando o Serial Monitor da IDE do Arduino.

Se não estiver funcionando como o esperado:

- Revise todas as ligações do circuito;
- A tensão da fonte de alimentação do Arduino;
- A ligação dos cabos do barramento RS485;

- Verifique se a porta COM do conversor USB é a que foi selecionada na IDE do Arduino;
- Se o baud rate no serial monitor está em 9600;
- Se o drive do FTD232RL foi corretamente instalado no gerenciador de dispositivos de hardware do PC.

Se ainda não estiver funcionando corretamente, entre em contato com o suporte da loja do LDG.

Teste com o SCADABR

Agora que já testamos a rede vamos mostrar um exemplo de como utilizar a solução open source ScadaBR para monitorar e controlar um Arduino configurado como escravo MODBUS usando a rede RS485 e o protocolo MODBUS padrão RTU, basta seguir a rotina abaixo.

O sketch que será usado no Arduino foi baseado em um exemplo disponível na biblioteca ModbusSlave de Juan Pablo Zometa que pode ser baixado em https://sites.google.com/site/jpmzometa/arduino-mbrt/arduino-modbus-slave

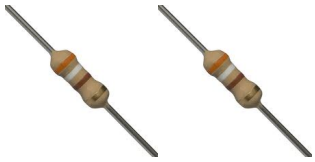
Mas para o exemplo a seguir não será necessário incluir a biblioteca na IDE da Arduino. Neste exemplo iremos monitorar o estado de uma chave táctil em uma entrada digital, o valor lido de uma entrada analógica e controlar o estado de um LED ligado em uma saída digital do Arduino. Siga os passos abaixo:

1. Para este exemplo vamos precisar de mais alguns componentes além dos que foram usados no teste inicial:

a. 1 LED de 3mm amarelo;

b. 1 LED de 3mm verde;

c. 2 resistores de 390Ω.



2. Instalar o shield ao Arduino e o conversor ao PC conforme mostrado nos itens instalação e configuração;
3. Montar o circuito nos conectores do shield usando os componentes conforme mostrado na figura 6.
4. ATENÇÃO: Revisar todas as ligações com muito cuidado antes de energizar o circuito, para evitar possíveis danos aos aparelhos!!!

5. Ligar uma fonte de alimentação de 7 a 12 VCC ao conector de alimentação externa do Arduino.

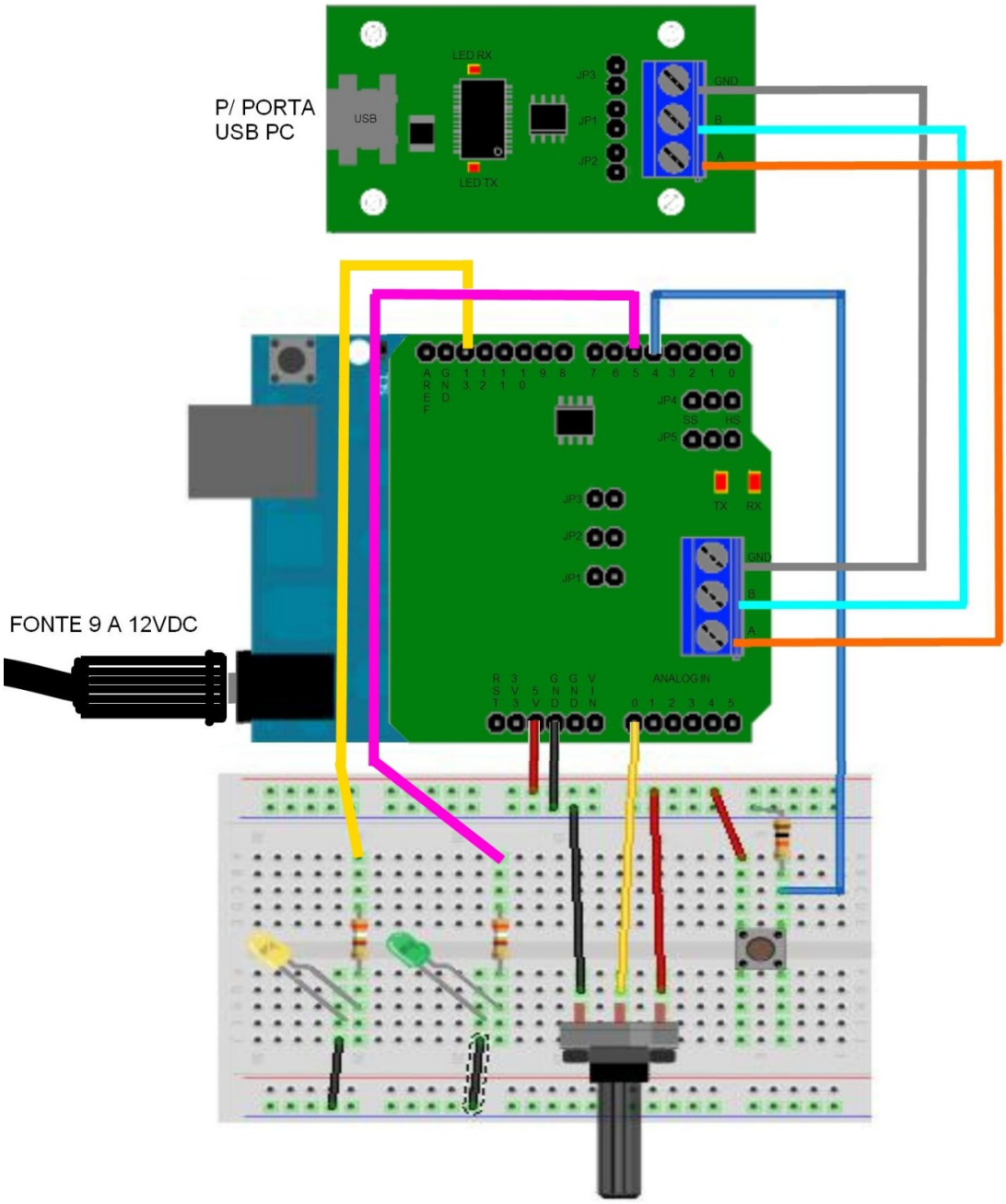


Fig. 6: Montagem para o exemplo com o SCADABR.

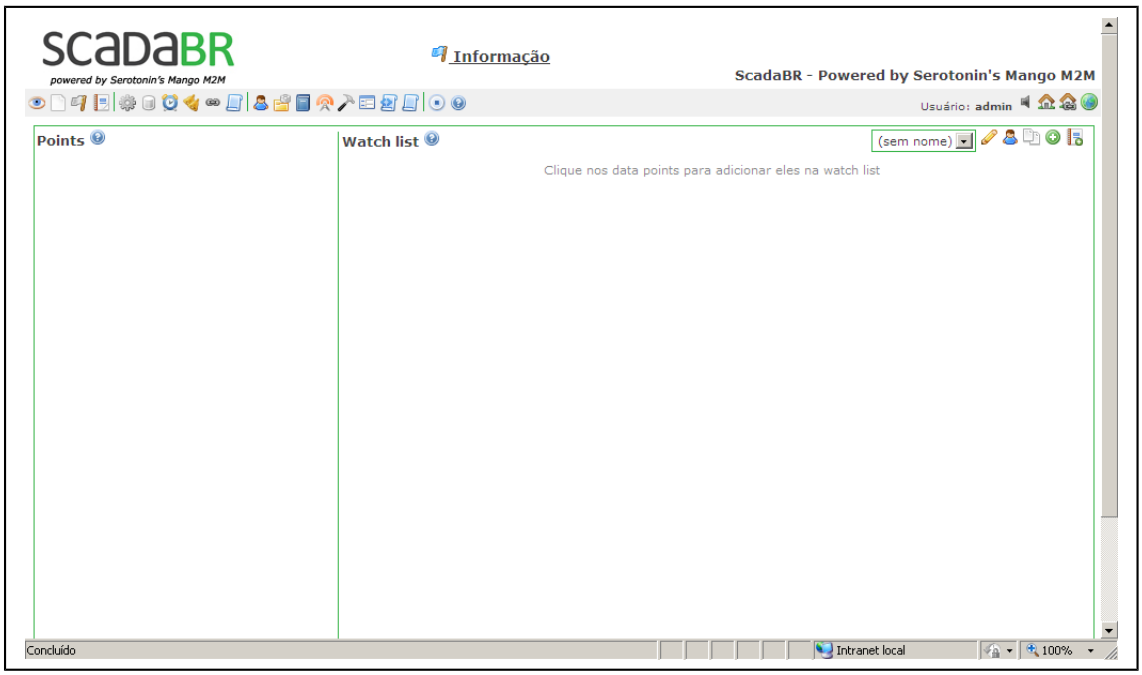
6. Abra um novo sketch na IDE e copie o código do apêndice IV.

7. Faça o upload para o Arduino conectando o cabo USB no conector do Arduino e a uma porta USB do PC. Não se esqueça de desligar os jumpers JP4 e JP5 durante o upload do sketch religando-os em seguida;

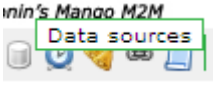
8. Assim que a mensagem confirmando que o upload terminou corretamente, desligue o cabo USB do Arduino e feche a IDE do Arduino e mantenha o conversor USB/RS485 conectado ao PC.

9. Para baixar e instalar o software SCADABR, siga as instruções contidas no manual no site: www.scadabr.org.br

10. Uma vez instalado o programa, rode a aplicação e faça o login, o default é admin para o User Id e para o Password. Uma página como essa deverá aparecer:



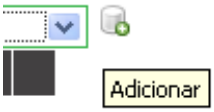
11. Inicialmente devemos criar uma data source, que identifica cada dispositivo na rede.

12. Clique em Data sources 

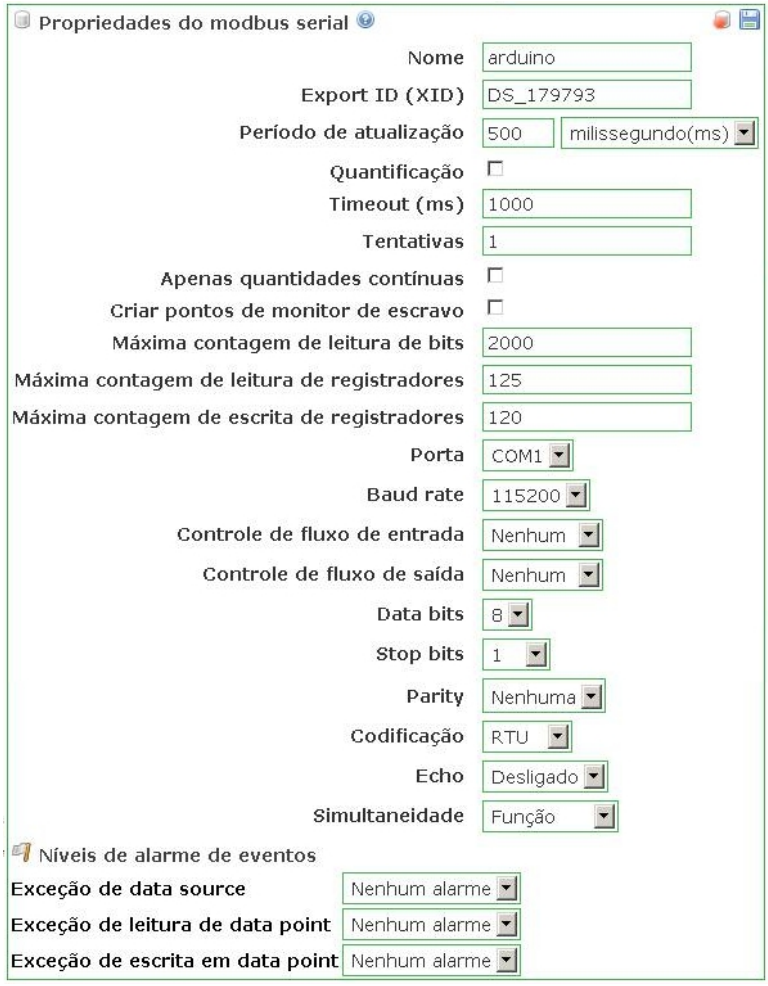
para abrir a lista de data source.

13. Clique em na aba e selecione Modbus Serial:

Rua Berta, 60 – Vila Mariana
São Paulo – SP – Brasil
(11) 3804-0126
Rede Social: http://www.labdegaragem.com.br
Loja Virtual: http://loja.labdegaragem.com.br

14. Clique em adicionar dispositivo: 

15. Abrirá uma tela parecida com a figura abaixo:



Preencha os seguintes dados:

Nome: Arduino

Periodo de Atualização: 1000ms

Porta: Use a COM que foi criada pelo conversor USB/RS485

Baudrate: 9600, deixar igual a linha do sketch COMM_BPS = 9600, /* baud rate */

Data Bits: 8

Stop Bits: 1

Parity: Nenhuma, deixar igual a linha do sketch PARITY = 'n', /* paridade */

Codificação: RTU

OBS: Verifique a versão do seu PC (versão 32bits ou 64bits) e instale o Java de acordo com esta versão.

As outras opções desta caixa não precisam ser alteradas.

Após preencher tudo, clique em salvar

16. Tendo criado o dispositivo de dados ou data source, devemos inserir os pontos de medição que são chamados de data points. Clique em para adicionar um data point no fim da página.
17. Para criar o data point que corresponde a leitura da chave ligada no pino digital D4 preencha os campos da janela aberta desta forma:

Preencha o campo nome de forma semelhante ao nome dado a respectiva variável no sketch do Arduino.

A Id do escravo, deve ser a mesma do sketch do Arduino:

MB_SLAVE = 1, /* endereço do escravo modbus.

Na biblioteca JPMZOMETA foram implementados o comandos 3, 6 e 16 do protocolo Modbus, que são os comandos para leitura e escrita dos registros do tipo holding. Desta forma o campo faixa do registro de todos os data source deverão estar marcados como registrador holding.

O campo tipo de dados modbus varia conforme o tipo da variável, para representar o estado de um pino digital, use binário e o Bit em 0. Para uma variável analógica marcar como inteiro de dois bytes com sinal.

O campo offset deve estar de acordo com a posição ocupada pela variável na tabela enum do sketch.
Marcar a caixa configurável apenas nas variáveis onde o SCADABR deve escrever, alterando o valor das mesmas no sketch do Arduino. As variáveis que representam entradas digitais e analógicas não devem ser marcadas.

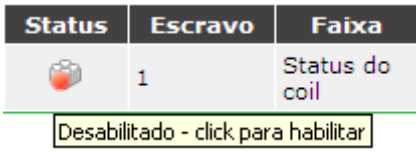
Após preencher tudo, clique em salvar o data point criado.

18. Para criar o data point que corresponde ao LED ligado no pino digital D5 preencha os campos da janela aberta desta forma:

19. Após preencher tudo, clique em para salvar o data point criado.
20. Para criar o data point que corresponde a leitura do potenciômetro ligado na entrada analógica A0 preencha os campos da janela aberta desta forma:

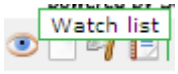
Rua Berta, 60 – Vila Mariana
São Paulo – SP – Brasil
(11) 3804-0126
Rede Social: http://www.labdegaragem.com.br
Loja Virtual: http://loja.labdegaragem.com.br

21. Após preencher tudo, clique em para salvar o data point criado.
22. Agora você deve habilitar todos os data points criados, clique no botão vermelho na na coluna Status de cada data point.

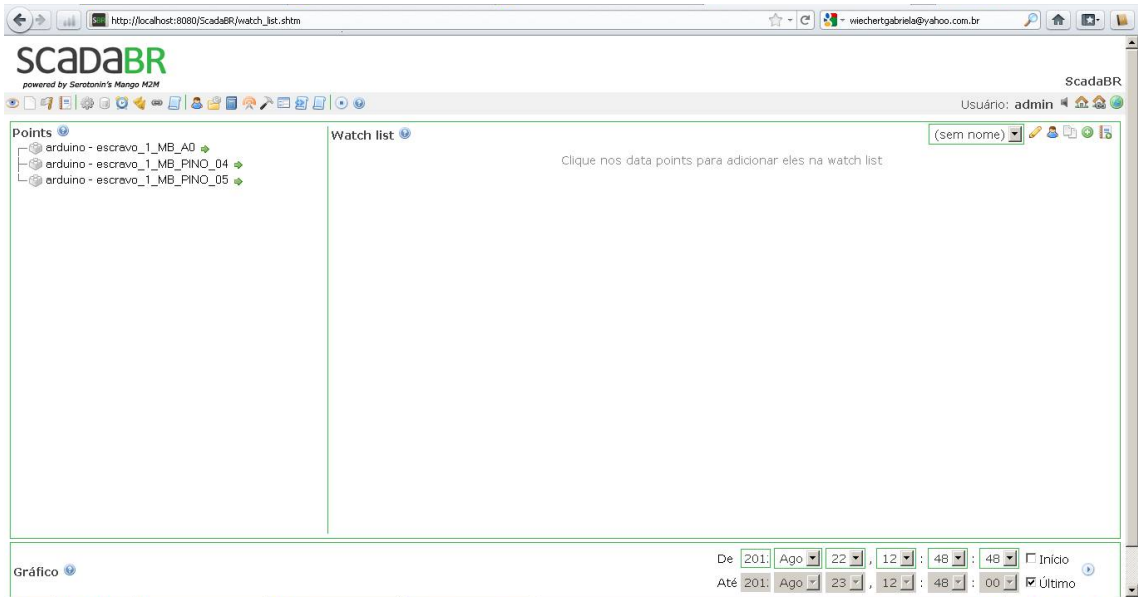


23. A caixa Data Points deve estar parecida com a figura abaixo:

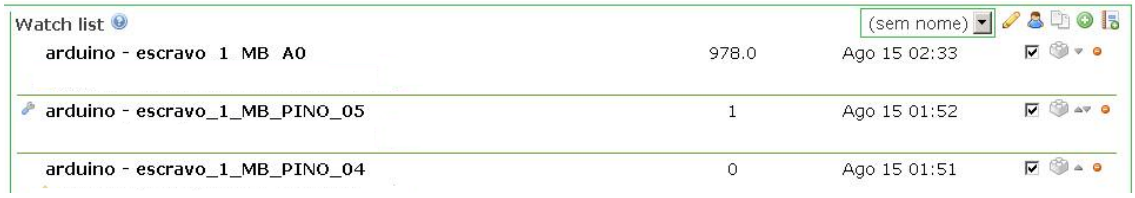
24. Na caixa Propriedades do modbus serial clique no botão vermelho para habilitar o Data Source.

25. Clicar no olho para acessar a tela Watch list 

26. A tela será como esta:



27. Clique nas setas dos data points que desejar monitorar, assim eles vão aparecer na caixa da direita da tela.

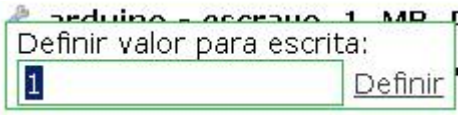


Com o Arduino ligado, o ScadaBR vai começar o polling atualizando os valores dos data points habilitados no intervalo de tempo definido no campo período de atualização na caixa propriedades do modbus serial.

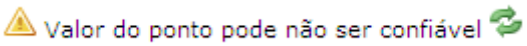
Variando o potenciômetro na entrada digital o valor do data point MB A0 será atualizado continuamente.

Quando o botão táctil estiver pressionado, o data point MB_PINO_4 estará em 1 e irá marcar 0 quando o botão não estiver pressionado. O LED amarelo ligado no pino digital 13 do Arduino acende sempre que o botão estiver pressionado.

O data point MB_PINO_5 controla o estado do LED verde ligado no pino digital 5. Clicando em será aberta uma caixa onde é possível ligar ou desligar o LED digitando 1 ou 0 respectivamente.



Se ocorrer um problema qualquer na comunicação, ou se o data source ou o data point não estiver habilitado uma mensagem de aviso como esta será mostrada.

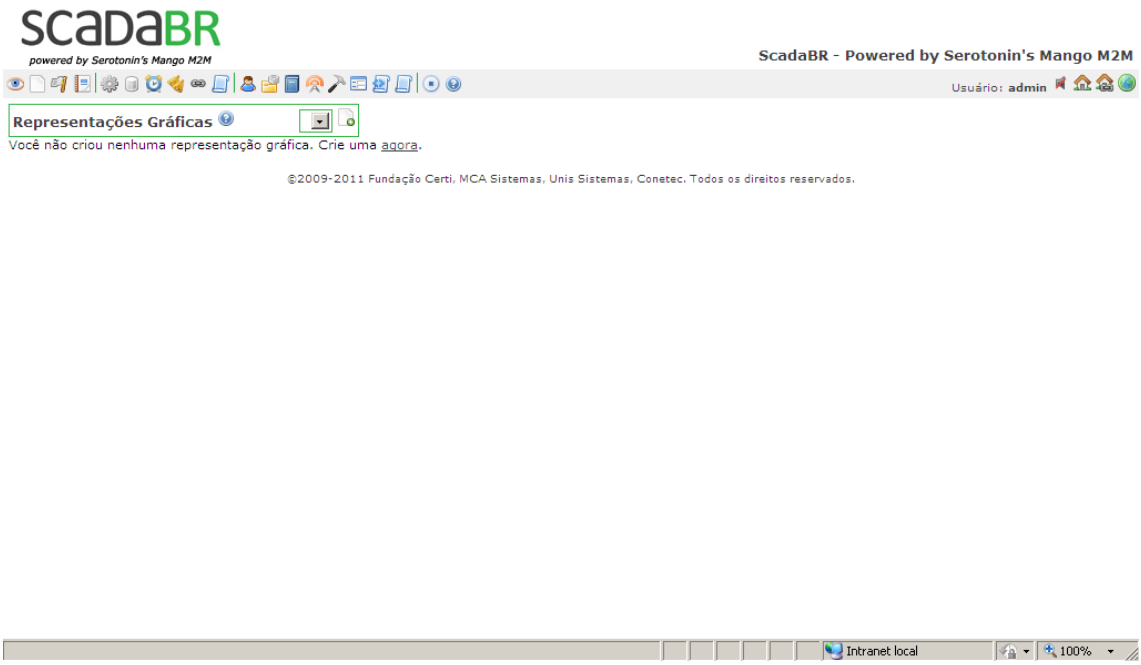


Representação Gráfica

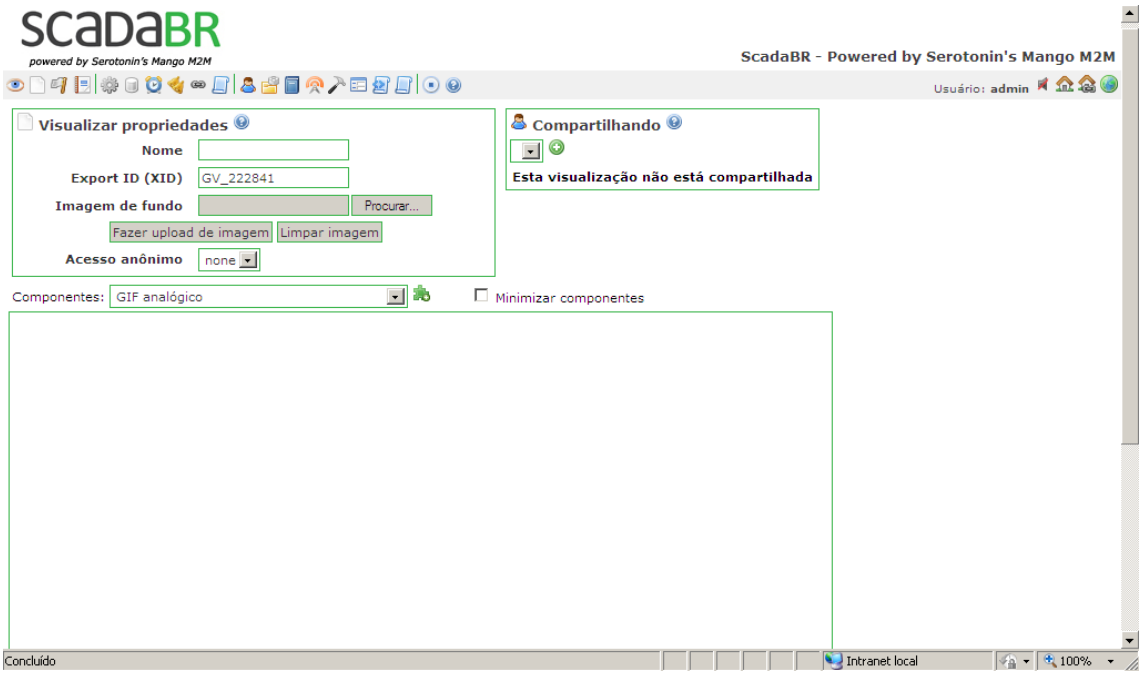
Agora que a aplicação já esta rodando no SCADABR, vamos aprender como fazer uma tela gráfica simples, de modo a ter uma visualização do sistema de uma forma mais fácil que do que a que temos na tela watch list.

Siga os passos abaixo:

1. Clique em para abrir a tela de representação gráfica.
2. Uma tela como a seguir deve aparecer:


ScadaBR
powered by Serotonin's Mango M2M
ScadaBR - Powered by Serotonin's Mango M2M
Usuário: admin
Representações Gráficas
Você não criou nenhuma representação gráfica. Crie uma agora.
©2009-2011 Fundação Certi, MCA Sistemas, Unis Sistemas, Conetec. Todos os direitos reservados.
Intranet local
100%

3. Clique em para criar uma nova tela de representação gráfica.
4. Uma tela como a seguir deve aparecer:

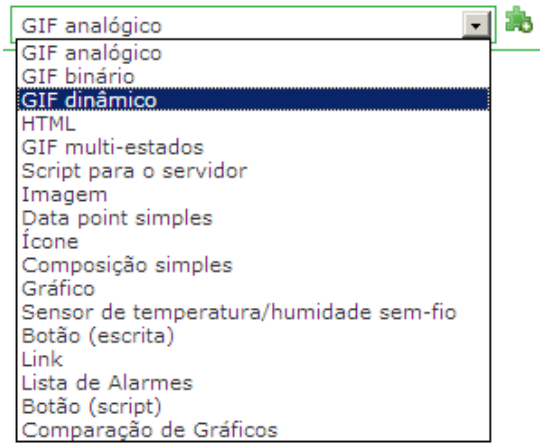


Preencha os campos a seguir:

Nome: Tela 1

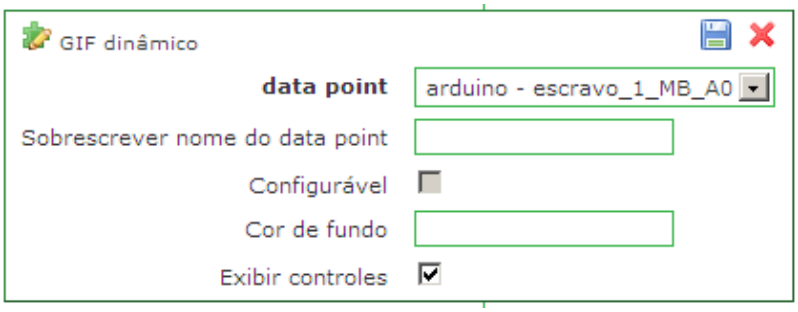
Se desejar carregar uma imagem para a tela de fundo, aponte o caminho em procurar e faça o upload da imagem. Para criar um elemento na tela, selecione o tipo do componente e clique em .

5. Para criar uma escala que represente o valor da leitura da entrada analógica A0, selecione GIF binário e clique em adicionar componente à representação.



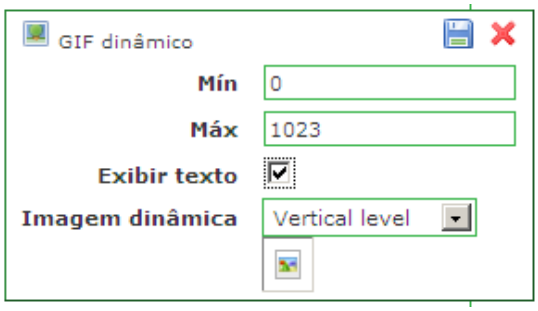
6. Deverá aparecer no centro da tela um objeto como este:

7. Passe o mouse sobre o objeto e clique sobre o ícone para configurar a propriedades do GIF, preenchendo os campos conforme a figura abaixo, salve as alterações.



8. Clique em para editar as propriedades gráficas do objeto e configure conforme a figura abaixo e salve as modificações:

Rua Berta, 60 – Vila Mariana
São Paulo – SP – Brasil
(11) 3804-0126
Rede Social: http://www.labdegaragem.com.br
Loja Virtual: http://loja.labdegaragem.com.br

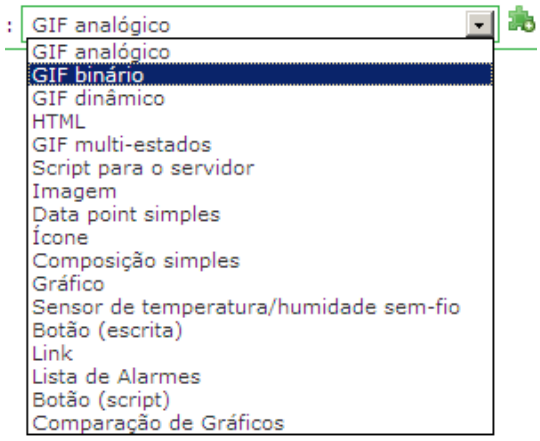


9. O objeto deve ter agora a seguinte aparência:

quando o potenciômetro estiver no mínimo;

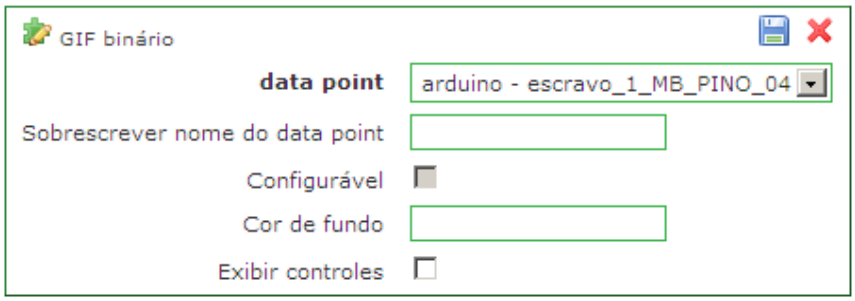
quando o potenciômetro estiver no máximo;

10. Para a representação do estado do botão táctil que está ligado no pino digital 4, vamos criar um LED amarelo na tela, para tanto, selecione GIF binário e clique em adicionar componente à representação.

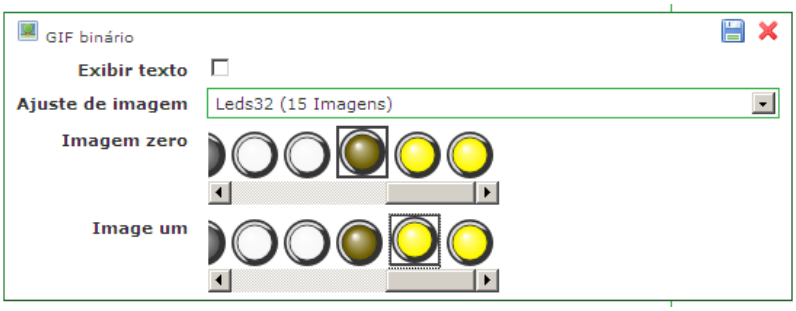


11. Deverá aparecer no centro da tela um objeto como este:

12. Passe o mouse sobre o objeto e clique sobre o ícone para configurar a propriedades do GIF, preenchendo os campos conforme a figura abaixo, salve as alterações.



13. Clique em para editar as propriedades gráficas do objeto e configure conforme a figura abaixo e salve as modificações:

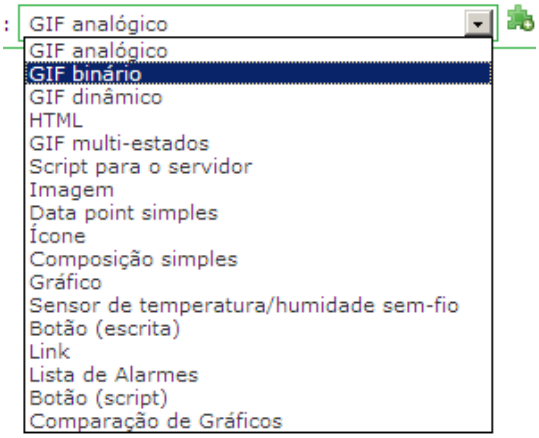


14. O objeto deve ter agora a seguinte aparência:

quando o botão não estiver pressionado;

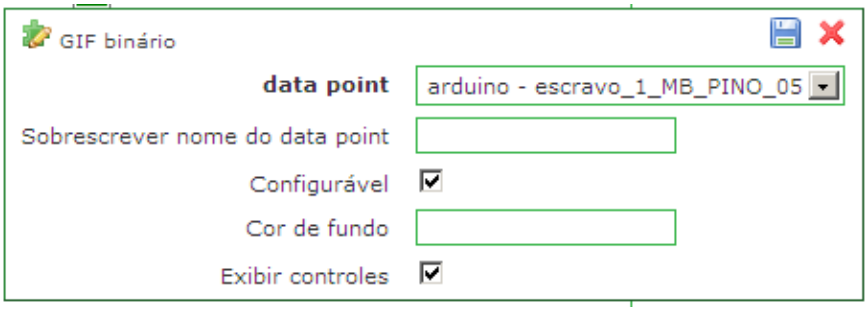
quando o botão não estiver pressionado.

15. Para a representação do acionamento da saída no pino digital 5, vamos criar um LED verde na tela, para tanto, selecione GIF binário e clique em adicionar componente à representação.

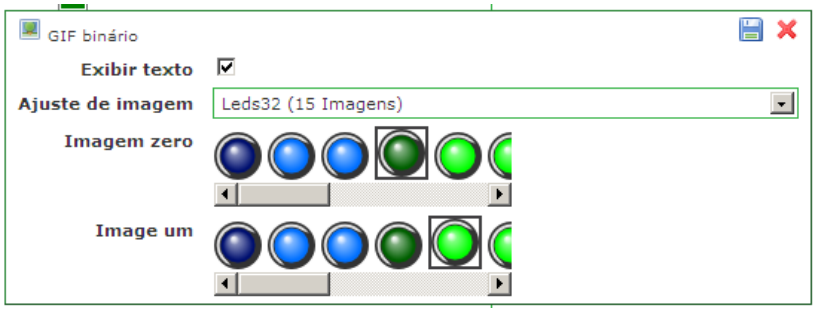


16. Deverá aparecer no centro da tela um objeto como este:

17. Passe o mouse sobre o objeto e clique sobre o ícone para configurar a propriedades do GIF, preenchendo os campos conforme a figura abaixo, salve as alterações.



18. Clique em para editar as propriedades gráficas do objeto e configure conforme a figura abaixo e salve as modificações:

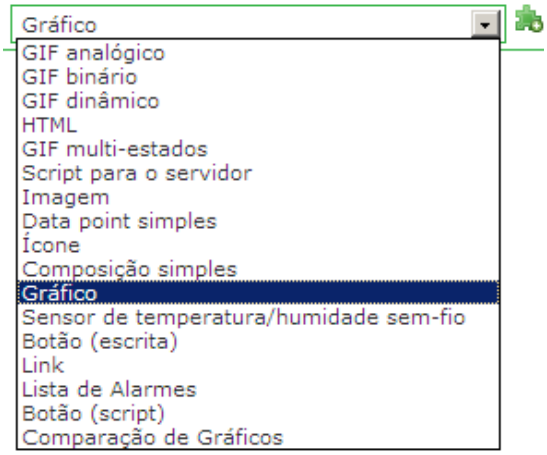


19. O objeto deve ter agora a seguinte aparência:

quando o botão não estiver pressionado;

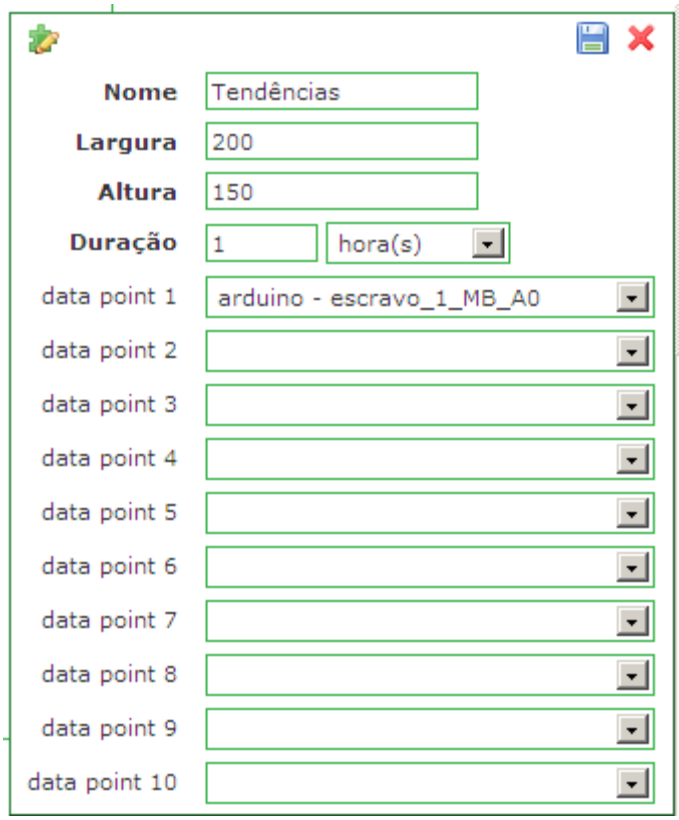
quando o botão não estiver pressionado.

20. Para criar um gráfico de tendências, selecione em gráfico e insira o componente:

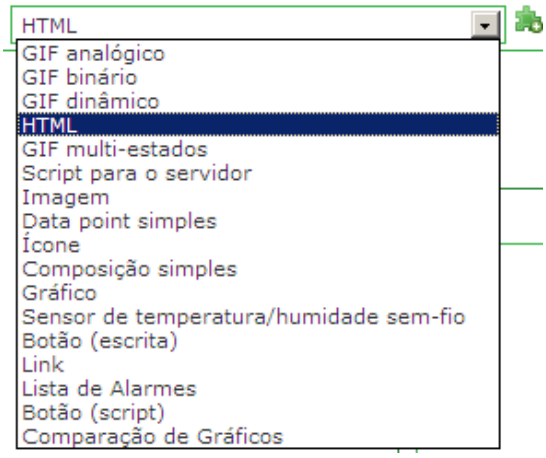


21. Deverá aparecer no centro da tela um objeto como este:

22. Clique em e preencha os campos conforme a figura abaixo, salve as alterações.



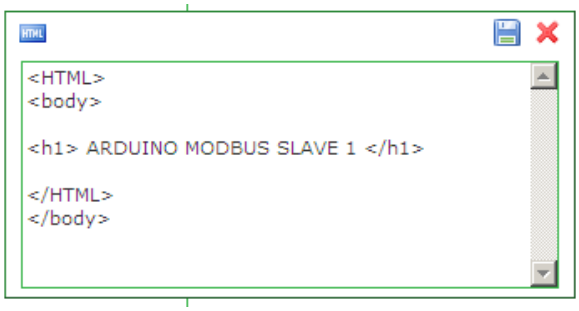
23. Para inserir textos estáticos na tela usamos a linguagem HTML, selecione HTML e insira um novo componente na tela:



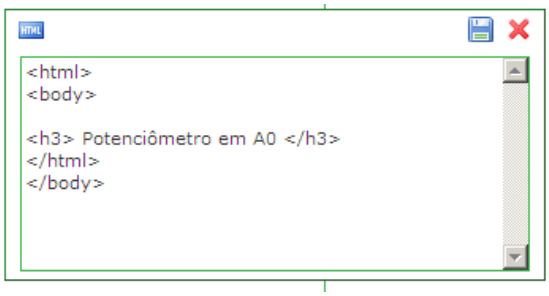
Rua Berta, 60 – Vila Mariana
São Paulo – SP – Brasil
(11) 3804-0126
Rede Social: http://www.labdegaragem.com.br
Loja Virtual: http://loja.labdegaragem.com.br

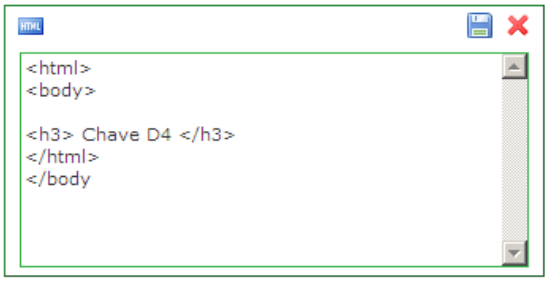
24. Deverá aparecer no centro da tela um objeto como este:

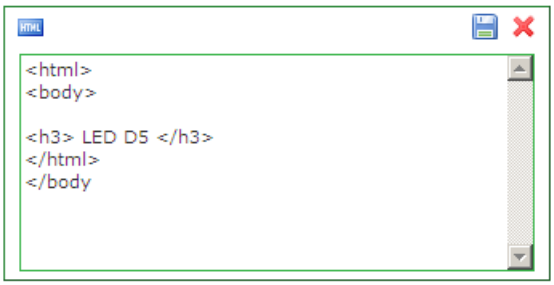
25. Passe o mouse sobre o objeto e clique sobre o ícone e digite o código em HTML conforme a figura abaixo, salve as alterações.



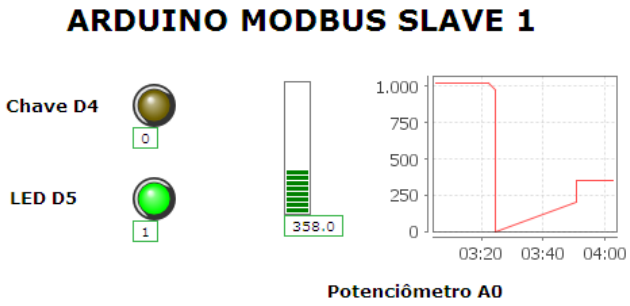
26. Crie um texto para identificar cada componente da tela:





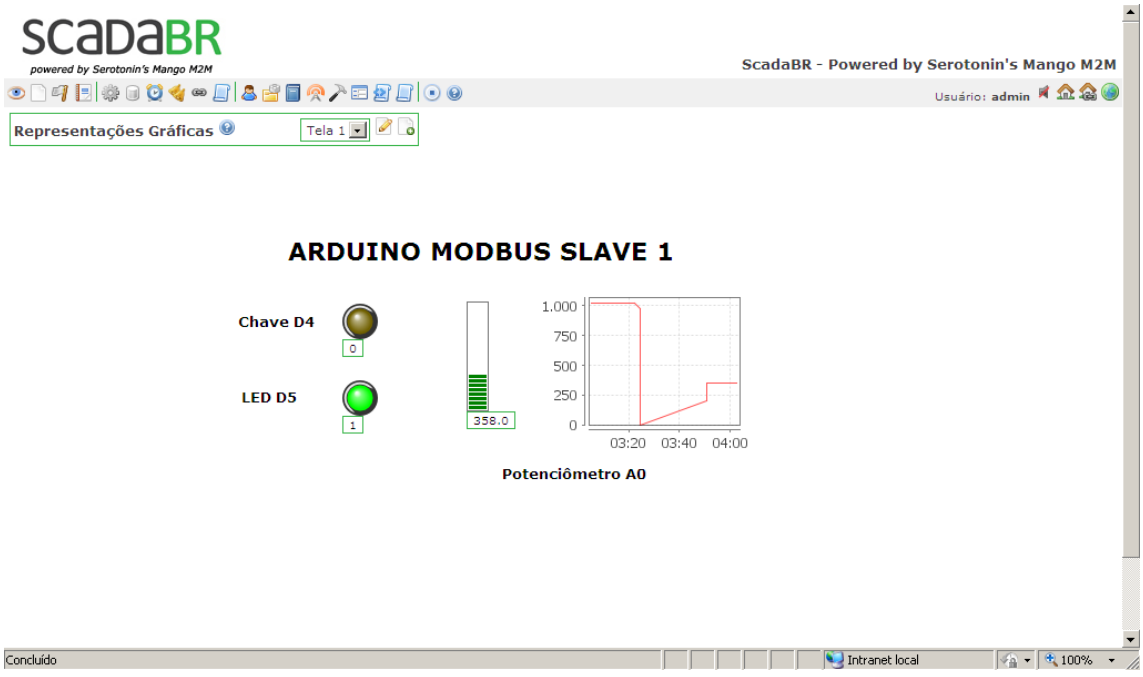


27. Araste os componentes alinhando conforme a figura abaixo:



28. Salve a tela.

29. A aparência final da tela:



30. Ao passar o mouse sobre os componentes são abertas caixas contendo as informações de cada um. Para mudar o estado do LED verde, passe o mouse sobre e digite o valor desejado, 0 para desligar ou 1 para ligar a saída, da mesma forma como é feito na tela watch list.

Apendice I:

Introdução ao barramento RS485

Antes de experimentar as vantagens do uso do KIT com a rede RS485, é preciso conhecer um pouco sobre as características e requisitos mínimos de um barramento RS485. Quando uma rede precisa transferir pequenos blocos de informação por longas distâncias, o RS485 é muitas vezes a interface mais indicada. Os nós da rede podem ser PCs, microcontroladores, ou quaisquer dispositivos capazes de comunicações seriais assíncronas. Comparado com Ethernet e outras interfaces de rede, o RS485 tem requisitos de hardware e de protocolo mais simples e mais baratos.

O documento de especificação, TIA-485-A, define as características elétricas da linha, seus drivers e receptores, mas não define a pinagem de conectores ou protocolos de comunicação por software.

O gerenciamento da rede é realizado pela adoção de um protocolo de comunicação comum a todos os dispositivos da rede. Este protocolo é um conjunto de regras que define como, quando e quem pode iniciar as transmissões na rede. Normalmente é usada uma configuração mestre-escravo (master-slave), onde um dispositivo é chamado de mestre e somente ele pode iniciar a comunicação com os outros elementos da rede (escravos). Um escravo apenas responde as solicitações enviadas pelo mestre da rede, nunca iniciando uma comunicação por conta própria.

Apesar do padrão RS485 não especificar nada sobre os protocolos, a maioria das redes RS485 usam os protocolos assíncronos suportados pelas UARTs dos PCs e outros computadores. Uma palavra transmitida consiste de um bit de início seguido por bits de dados, um bit de paridade opcional, e um bit de finalização.

Uma rede RS485 pode ter até 32 dispositivos, com uma carga equivalente a uma impedância de entrada de 12k ohms cada.

O meio físico mais utilizado é um cabo de par trançado. É através deste único par de fios, chamado de barramento, e por onde cada dispositivo transmite e recebem os dados. **Cada dispositivo aciona o seu transmissor apenas no instante que necessita transmitir, mantendo-o desligado no resto do tempo de modo a permitir que outros dispositivos transmitam dados**. Em um determinado instante de tempo, somente um dispositivo pode transmitir, o que caracteriza esta rede como half-duplex. **No shield essa função de comutar entre o modo de recepção e transmissão é feita pelo pino digital 6 que no Arduino deve estar configurado como saída, sendo que para recepção o pino deve estar em LOW e em HIGH para poder transmitir. No conversor USB a comutação entre o modo de transmissão e recepção é feita de forma automática pelo FT232RL**.

Um barramento RS485 pode estender-se até o comprimento máximo de 1200 metros (4000 pés), e podem transferir dados a uma taxa de até 10 Mbps, mas não ambos ao mesmo tempo. Com 90 kbps o comprimento máximo é de 1200 m, para uma taxa de 1 Mbps, a distância cai para 120m, e 10 Mbps para 15m. Para usar mais nós ou para distâncias muito mais longas, você pode usar repetidores que regeneram os sinais e começam um novo barramento RS485. A figura 7 ilustra a relação entre a taxa de transmissão e a distância de um barramento.

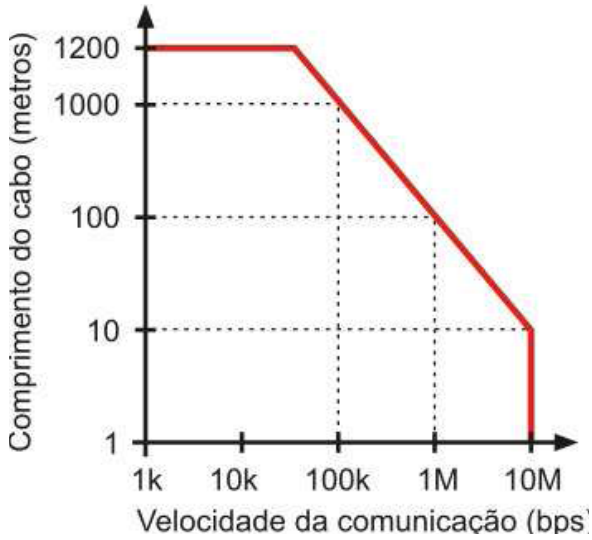


Fig. 7: Gráfico Comprimento x Velocidade

No caso do nosso Kit, a maior taxa de comunicação está restringida a no máximo 115.2 kbps devido à limitação no chip conversor USB FT232RL.

A principal razão para que o barramento RS-485 consiga se estender a essas distâncias é o uso de sinais não balanceados, ou diferenciais. Dois fios, geralmente um par trançado, um destes fios leva o sinal de tensão normal e o outro fio leva o seu inverso. O receptor detecta a diferença entre os dois. Como o ruído captado pelo cabo da linha de transmissão é comum a ambos os fios os fios, ele acaba por ser cancelado no receptor diferencial.

Os manuais dos chips costumam chamar o terminal do sinal não invertido do barramento RS-485 de “A” e a linha invertida como “B”. Um receptor RS-485 deve ser capaz de detectar uma diferença de tensão de pelo menos 200 mV entre A e B. Se A é, pelo menos, 200 mV maior do que B, a saída do receptor é um nível lógico alto (1). Se B é, pelo menos, 200 mV maior do que A, a saída é um nível lógico baixo (0). Para diferenças menores que 200 mV, a saída é indefinida a figura 8 mostra estes sinais.

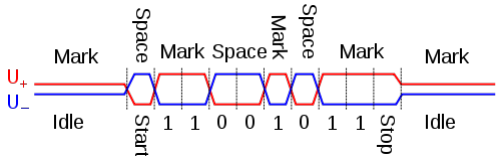


Fig. 8: Conexão dos dispositivos ao barramento.

O RS-485 foi projetado para ser conectado preferencialmente em uma topologia de barramento ou Daisy Chain. A maioria dos links utiliza pares trançados devido à sua capacidade para cancelar magneticamente o ruído eletromagnético acoplado. As figuras 9 e 10 mostram as formas de ligação possíveis dos dispositivos ao barramento. Note que os terminais “A” de todos os dispositivos da rede estão ligados ao mesmo fio da linha de transmissão, e que todos os terminais “B” estão ligados ao outro fio da linha de transmissão.

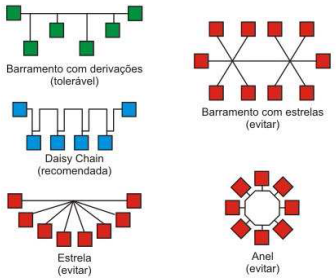


Fig. 9: Conexão dos dispositivos ao barramento.

Em linhas de transmissão de uma rede de comunicação sempre existe a necessidade de se fazer a terminação das com um valor de impedância correspondente à impedância característica da linha de transmissão. A correta terminação atenua reflexões que distorcem os dados transmitidos, aumentando os limites de velocidade e/ou comprimento da rede. Redes com cabos curtos (até 100 m) e operando a baixa velocidade (até 19200 bps) operam adequadamente mesmo sem a utilização de resistores de terminação. Os resistores devem possuir um valor igual à impedância característica da linha, ou seja, em torno de 120 ohms e um resistor deve ser posicionado em cada uma das extremidades da rede. Em uma das extremidades do barramento temos também dois resistores de 560 ohms de polarização, um ligado ao VCC e outro ao GND. Estas resistências são chamadas de pull up e pull down e servem para polarizar a linha de transmissão de forma a estabilizar a mesma contra as reflexões de tensão que podem causar erros no receptor.

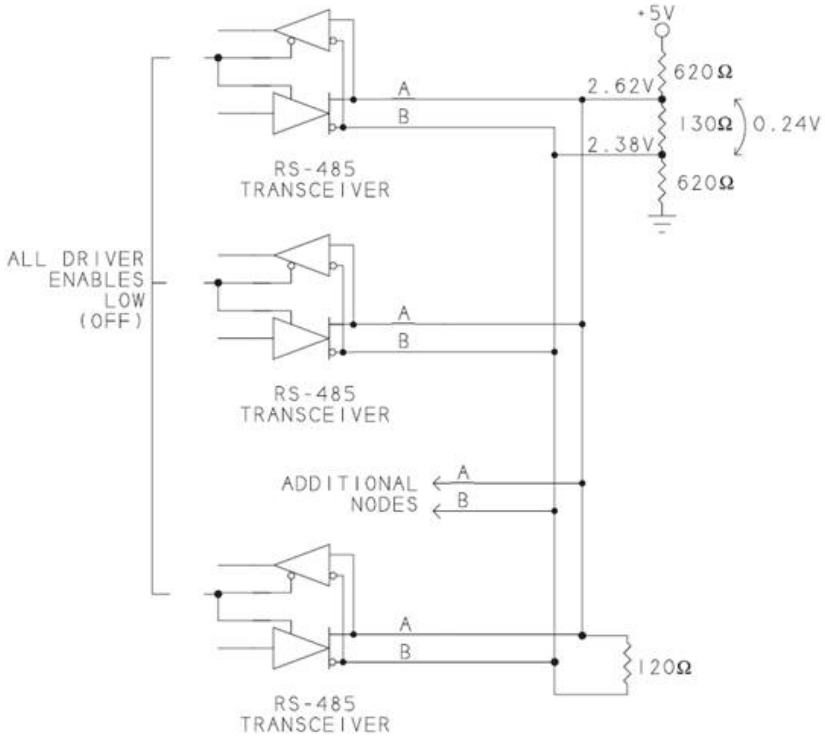


Fig. 10: Conexão detalhada dos dispositivos ao barramento.

Apendice II:

Introdução ao protocolo MODBUS padrão RTU

O Protocolo MODBUS foi criado no final dos anos 70 para comunicação entre controladores da MODICON. Por ser um dos primeiros protocolos com especificação aberta foi adotado em vários níveis de aplicações, inclusive sistemas de supervisão, tornou-se um padrão mundial devido a sua simplicidade e facilidade de interligar equipamentos de fabricantes diferentes. O MODBUS é o protocolo de mais larga utilização em automação industrial até hoje

O Protocolo define como o dispositivo Mestre solicita o acesso a outro dispositivo Escravo na rede, como o escravo responderá as solicitações do mestre, como os erros serão detectados e informados e os tempos de chamada e de resposta das mensagens.
A arquitetura usada é mestre-escravo. Isto é, o somente o mestre pode iniciar a comunicação e o escravo chamado pelo mestre apenas responde as solicitações do mestre, nunca iniciando uma chamada. O mestre pode ser um CLP (Controlador Lógico Programável) ou um computador operando um SCADA (Sistema de supervisão). Os escravos podem ser CLP's, computadores, sensores inteligentes etc.
Quanto a topologia física da rede pode ser Ponto a Ponto com RS-232, barramento Mutiponto com RS-485 ou TCP/IP sobre ethernet.

Neste artigo será discutido apenas o padrão RTU que é na prática o mais difundido em linha serial.

Mensagens MODBUS - RTU

As mensagens em MODBUS são formadas por frames ou quadros e existem apenas dois tipos chamados de quadro de consulta e quadro de resposta. A figura 11 ilustra uma transação entre o mestre e o escravo sendo que o quadro da esquerda representa o frame de solicitação enviado pelo mestre ao escravo e o quadro da direita é frame de resposta enviado ao mestre pelo escravo.

Figura 11: Ciclo de mensagens no protocolo MODBUS.

Quadro de consulta (solicitação do mestre)

No quadro de consulta, o código de função informa ao dispositivo escravo com o respectivo endereço, qual a ação a ser executada. Os bytes de dados contêm informações para o escravo, por exemplo, qual o registrador inicial e a quantidade de registros a serem lidos. O campo de verificação de erro permite ao escravo validar os dados recebidos e determinar se ocorreu um erro na transmissão da mensagem.

Start - equivale a 3,5 vezes o tempo de transmissão de um byte conforme configuração da porta serial (número de start bits + número de bits por caracter + bit de paridade + número de stop bits).
Adress – formada por um byte que representa o endereço de destino da mensagem, indica com qual escravo o mestre deseja iniciar a transação. Pela norma os escravos podem ter endereços de 1 a 247, o endereço 0 é reservado para mensagens de broadcast, isto é, todos os escravos irão reconhecer a mensagem e executar o comando do mestre, neste tipo específico de mensagem os escravos não enviam mensagem de retorno para o mestre.
Function – formada por um byte que representa o código da função ou comando solicitado, indicando oque este frame faz.
Data – pacote de dados formado por um número de bytes variável conforme o comando do frame.

CRC – composto por dois bytes de verificação de erros, onde é feita a checagem dos dados enviados a fim de se verificar se ocorreu um erro durante a transmissão do frame.
Consiste em realizar uma operação de divisão sobre o bloco de dados com um divisor conhecido, utilizando-se o resto da operação como um caracter de verificação. A divisão é feita considerando-se como o dividendo a seqüência de bits que será transmitida, sendo que o divisor é convenientemente escolhido de modo a controlar o tamanho em bits do resto da divisão. É muito comum a utilização de CRC de no máximo 16 bits, o que para a maior parte dos casos fornece uma boa segurança.

Quadro de resposta (resposta do escravo)

No quadro de resposta, o código de função é repetido de volta para o mestre. Os bytes de dados contêm os dados coletados pelo escravo ou o seu estado. Se um erro ocorre, o código de função é modificado para indicar que a resposta é uma resposta de erro e os bytes de dados contém um código que descreverá o erro. A verificação de erro permite o mestre validar os dados recebidos. Se o escravo não responder dentro de um tempo pré determinado o mestre entende que foi gerado um erro de time out e tomará as providências programadas.

Adress – formada por um byte que representa o endereço do escravo que está respondendo ao mestre.
Function – formada por um byte que representa o código da função ou comando solicitado pelo mestre.
Data – pacote de dados formado por um número de bytes variável conforme o comando do frame.
CRC – composto por dois bytes de verificação de erros, onde é feita a checagem dos dados enviados a fim de se verificar se ocorreu um erro durante a transmissão do frame.

Duração do Frame RTU

Na figura 12 vemos a formação dos quadros do tipo RTU. Observe primeiramente que temos um período de silêncio, onde sempre que as linhas de comunicação ficarem sem tráfego por pelo menos 3.5 caracteres o escravo entende que a rede entrou em estado de idle, ou seja, em estado de repouso. Para uma rede com taxa de comunicação 8N1 a 9600bps temos que o tempo de transmissão de 1 bit será de:

1/9600 = 104us

Como 8N1 representa 1 bit de start, 8 bits de dados, nenhum bit de paridade e 1 bit de parada temos ao todo 10 bits por caractere transmitido. Agora é só multiplicar o tempo de transmissão de um bit (104us) pelo número de bits do caractere para encontrar o tempo de transmissão de um caractere ou byte, neste caso:

104 x 10 = 1040us ou 1,04ms

Caso a rede fique sem transmitir por:

3,5 x 1.04ms = 3,64ms

Ou seja por se a rede ficar em silencio por 3,5 caracteres temos o estado de idle e a possibilidade do início da transmissão de um novo frame.

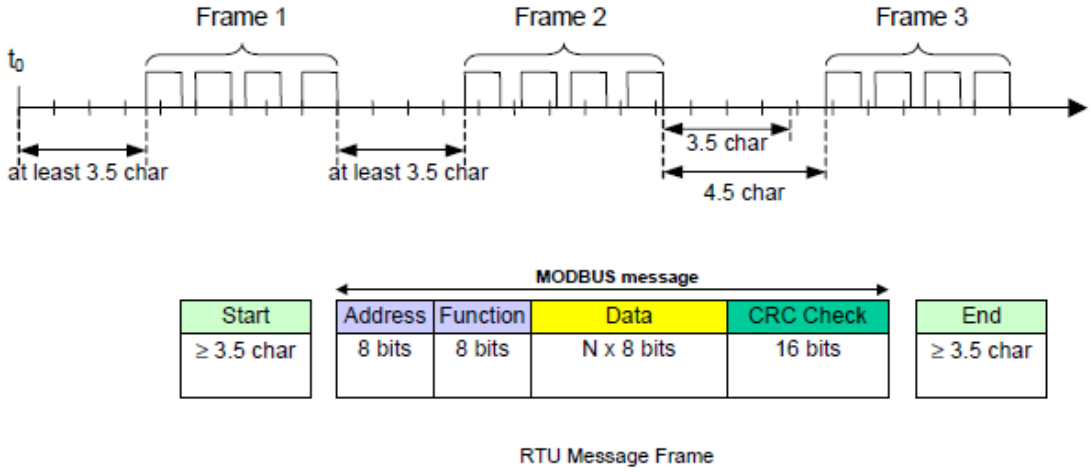


Figura 12: Tempos e composição dos frames no protocolo MODBUS-RTU.

Observe que o Start não faz parte do frame da mensagem
O primeiro campo de qualquer frame é composto por 8 bits que representam o endereço do escravo.
O segundo campo também é de 8 bits e informa a função ou o comando solicitado no frame. Os principais comandos do protocolo MODBUS estão listados mais abaixo.
O terceiro campo são os dados solicitados em um frame de solicitação ou os dados reportados pelo escravo no caso de um frame de resposta. Os dados tipicamente incluem registradores de endereços, contadores de valores e escrita de dados. A quantidade de bytes deste campo varia com o comando e quantidade de registros solicitados pelo mestre.
Finalmente temos o frame de CRC formado por 16 bits onde é feita a checagem de erros de transmissão.
A mensagem inteira deve ser transmitida continuamente. Se um intervalo de silêncio de 1,5 caractere ocorrer antes de completar o quadro, o dispositivo considera a mensagem incompleta e considera o próximo byte como o endereço da nova mensagem. Em um caso similar, se a nova mensagem iniciar 3,5 caractere antes do início da nova mensagem, o dispositivo receptor assume que ele está continuando com a mensagem prévia. Isto irá gerar uma mensagem de erro, assim como o valor final do campo CRC não será válido para combinar a mensagem.

Códigos das Funções

A identificação dos comandos (funções) de leitura e escrita são diferentes de acordo com o tipo de dado a ser lido ou escrito. As principais funções no protocolo MODBUS para troca de mensagens são:
• 1 - Leitura de um número variável de saídas digitais (bobinas);
• 2 - Leitura de um número variável de entradas digitais.
• 3 - Leitura de um número variável de registradores de memória.
• 4 - Leitura de um número variável de entradas analógicas.
• 5 - Escrita de um valor em uma única saída digital;
• 6 - Escrita de um valor em um único registrador de memória.

• 7 - Leitura de exceções (registros de erros);
• 8 - Escrita de valores em um número variável de saídas discretas;
• 15- Escrita de valores em um número variável de saídas digitais;
• 16- Escrita de valores em um número variável de registradores de memória.

Observem que alguns destes comandos requisitam a leitura ou escrita de vários operandos consecutivos. Nestes casos, no frame de consulta o mestre especifica o endereço do primeiro operando e o número de operandos envolvidos na comunicação.

Modelagem de Dados

Todo dispositivo em uma rede Modbus deve ter a sua memória dividida em registradores de 16 bits. A divisão é baseada na estrutura de memória de um CLP:

Coils - Saídas discretas para os atuadores ON-OFF utilizam um bit. Cada registrador comporta 16 saídas.
Inputs - Entradas discretas para os sensores ON-OFF utilizam um bit. Cada registrador comporta 16 entradas.
Input Registers - Entradas analógicas utilizam registradores de 16 bits para os valores obtidos por conversores A/D a partir dos sinais dos sensores analógicos.
Holding Registers - Registradores de Memória com 16 bits para os valores utilizados internamente no CLP.

Endereços dos Registradores	Memória Dispositivo	Comandos Leitura	Comandos Escrita
0XXXX	Solenóides Saidas Discretas	1	5,15
1XXXX	Entradas Digitais	2	N/D
3XXXX	Entradas Analógicas	4	N/D
4XXXX	Registradores de Memória	3	6,16

Figura 13: Endereços de memória reservados aos registradores no protocolo MODBUS-RTU.

Existe uma faixa de endereços destinada aos operandos de cada tipo de dados, conforme relacionados a seguir:

00001 a 09999 - Coils;
10001 a 19999 - Inputs;

30001 a 39999 - Input Registers;
40001 a 49999 - Holding Registers.

O endereço dos dados dentro das mensagens difere levemente do seu endereçamento lógico. Dentro das mensagens, o endereço dos operandos varia de 0 a 9998, independentemente do tipo de dado. No caso de um Input Register, o endereço lógico 30002 corresponde ao endereço 1 dentro das mensagens, o endereço lógico 30001 corresponde ao endereço 0 e assim por diante. O mesmo vale para os outros tipos de registros. O Holding Register 40001 equivale ao endereço 0 das mensagens referentes a Holding Registers. A princípio pode-se imaginar que isto cause certa confusão, pois não há como saber se o endereço 0 é um input, coil ou holding register, porém o tipo de registrador já está embutido no código da função.
Fica evidente a importância de compreender perfeitamente a tabela de alocação dos tipos de registros e endereços de um escravo no protocolo MODBUS. Pois o mestre se referencia pelos endereços através de dos comandos e da posição na tabela.
Todo dispositivo escravo MODBUS deve ter uma tabela com os registradores usados por este dispositivo, identificando o endereço, tipo de acesso (leitura e/ou escrita), descrição faixa de valores de forma a facilitar vida de quem for fazer a implementação do sistema de supervisão ou do mestre da rede. Abaixo é mostrado o exemplo de duas tabelas de registros de um determinado escravo. A tabela 1 mostra os registros de saídas digitais detalhando cada registro e a tabela 2 mostra a tabela de holding registers do mesmo dispositivo.

Dispositivo A

Tabela coils				
Endereço	**R/W**	**Descrição**	**Valor**	
0000_H	R/W	Habilitar função 50	Sim	1
0001_H	R/W	Habilitar função 51	Sim	1
0002_H	R/W	Habilitar função 50N	Não	0
0003_H	R/W	Habilitar função 51N	Não	0
0010_H	R/W	Contato 1	Fechado	1
0011_H	R/W	Contato 2	Aberto	0
0012_H	R/W	Contato 3	Aberto	0
0030_H	R	Atuação função 50	Sim	1
0031_H	R	Atuação função 51	Não	0
0032_H	R	Atuação função 50N	Não	0
0033_H	R	Atuação função 51N	Não	0

Tabela 1: Exemplo de endereços e descrição dos registros de saída de um dispositivo MODBUS-RTU.

Sempre que se for desenvolver um dispositivo que deve ser compatível com o protocolo MODBUS deve-se respeitar a distribuição de memória padrão estabelecida pelo protocolo. Porém fica claro que os registros do tipo holding registers podem ser considerados praticamente “coringas” e na prática verifica-se que os fabricantes de equipamentos são muito criativos no uso destes registros.

Tabela Registros				
Endereço	**R/W**	**Descrição**	**Unit**	**Valor**
0000_H	R	Corrente fase A	0,1A	2,1A
0001_H	R	Corrente fase B	0,1A	3,7A
0002_H	R	Corrente fase C	0,1A	6,1A
0003_H	R	Corrente neutro	0,1A	0,7A
0020_H	R/W	Set point função 50	0,1A	5,0A
0021_H	R/W	Set point função 51	0,1A	2,0A
0022_H	R/W	Set point função 50N	0,1A	3,0A
0023_H	R/W	Set point funcao 51N	0,1A	1,0A
0030_H	R/W	Tempo função 51	1 s	4 s
0031_H	R/W	Tempo função 51N	1 s	2 s

Tabela 2: Exemplo de endereços e descrição dos holding registers de um dispositivo MODBUS-RTU.

Conclusão

O protocolo Modbus é bastante eficiente e confiável, sendo uma maneira relativamente simples e fácil de implementar a comunicação ou troca de dados entre dois ou mais dispositivos em sistemas de automação. Por ser um dos primeiros protocolos abertos e não ser muito exigente no que se refere aos meios físicos de transmissão é um dos protocolos com menor custo de implantação e ainda hoje, mais de 30 anos após o lançamento continua sendo um dos protocolos de maior aceitação no mercado mundial.

Rua Berta, 60 – Vila Mariana
São Paulo – SP – Brasil
(11) 3804-0126
Rede Social: http://www.labdegaragem.com.br
Loja Virtual: http://loja.labdegaragem.com.br

Apendice III:

Sketch para teste de comunicação RS485 com o Serial Monitor ou Hyper Terminal

```
/*
Teste de Comunicação Serial com RS485
Este exemplo é de domínio público
Testado na IDE 1.0.1
*/

int POTENCIOMETRO = 0;    // valor lido no potenciometro na entrada analogica A0
int CHAVE = 0;            // Estado do push botton ligado na entrada D4
int RECEBIDO = 0;         // Byte recebido pela serial

void setup()
{
```

```
  // Iniciando a comunicação a 9600 bps:
  Serial.begin(9600);
  pinMode(4, INPUT);    // Chave no pino digital 4
  pinMode(6, OUTPUT); // TX ENABLE no pino digital 6 (está no SHIELD)
}

void loop()
{
  // Verifica se recebeu algum caracter pela serial
  if (Serial.available() > 0) {
    // le o byte recebido:
    RECEBIDO = Serial.read();
    // le a entrada analógica e divide por 4 para caber em um byte (0-255):
    POTENCIOMETRO = analogRead(A0)/4;
    // le o estado do botão:
    CHAVE = map(digitalRead(4), 0, 1, 0, 1);
    // envia os valores dos sensores:
    digitalWrite(6, HIGH);   // coloca o Shield RS485 em modo de transmissão
    delay(1);               // aguarda 1 milisegundo
    Serial.write("Pino A0 = ");
    Serial.print(POTENCIOMETRO);//Serial.println(POTENCIOMETRO, DEC);
    Serial.write('\n');
    Serial.write("Pino D4 = ");
    Serial.print(CHAVE);
    Serial.write('\n');
```

```
   Serial.write("Caracter recebido = ");

                Serial.write((byte)RECEBIDO);//Serial.write(RECEBIDO);//Serial.println(RECEBIDO,
BYTE);//Serial.write(byte(RECEBIDO));

   Serial.write('\n');

   delay(100);

   digitalWrite(6, LOW);   // coloca o Shield RS485 em modo de recepção

   Serial.flush();

   delay(300);            // aguarda

  }

}
```

Apendice IV:

Sketch para teste com o SCADABR e Arduino escravo MODBUS

```
/*
 Modbus serial - RTU Slave Arduino Sketch

 Este exemplo é de domínio público

 Testado na IDE 1.0.1

 Baseado na biblioteca de Juan Pablo Zometa : jpmzometa@gmail.com

 http://sites.google.com/site/jpmzometa/
```

and Samuel Marco: sammarcoarmengol@gmail.com

and Andras Tucsni.

As funções do protocolo MODBUS implementadas neste código:

3 - Read holding registers;

6 - Preset single register;

16 - Preset multiple registers.

This program is free software; you can redistribute it and/or modify

it under the terms of the GNU General Public License as published by

the Free Software Foundation; either version 2 of the License, or

(at your option) any later version.

This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU General Public License for more details.

You should have received a copy of the GNU General Public License

along with this program; if not, write to the Free Software

Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

The functions included here have been derived from the

Modicon Modbus Protocol Reference Guide

which can be obtained from Schneider at www.schneiderautomation.com.

This code has its origins with

```
 paul@pmcrae.freeserve.co.uk (http://www.pmcrae.freeserve.co.uk)
 who wrote a small program to read 100 registers from a modbus slave.

 */


/*
 * configure_mb_slave(baud, parity, tx_en_pin)
 *
 * configuração dos parametros da porta serial.
 *
 * baud: taxa de transmissão em bps (valores típicos entre 9600, 19200... 115200)
 * parity: seta o modo de paridade:
 *         'n' sem paridade (8N1); 'e' paridede impar (8E1), 'o' paridade par (8O1).
 * tx_en_pin: pino do arduino que controla a transmissão/recepção em uma linha RS485.
 *        0 or 1 desliga esta função (para rede RS232)
 *        >2 para uma rede multiponto.
 */
void configure_mb_slave(long baud, char parity, char txenpin);


/*
 * update_mb_slave(slave_id, holding_regs_array, number_of_regs)
 *
 * verifica se há qualquer pedido válido do mestre modbus. Se houver,
 * executa a ação solicitada
 *
 * slave: endereço do escravo (arduino) (1 to 127)
```

```
 * regs: uma matriz com os holding registers. Eles começam no endereço 1 (mestre ponto de vista)

 * Regs_size: número total de holding registers.

 * Retorna: 0 se não houver pedido do mestre,

 * NO_REPLY (-1) se nenhuma resposta é enviada para o mestre

 * Caso um código de exceção (1 a 4) em algumas exceções de modbus

 * O número de bytes enviados como resposta (> 4) se OK.

 */


int update_mb_slave(unsigned char slave, int *regs,

unsigned int regs_size);


/* Aqui começa o código do exemplo */


/* Parâmetros Modbus RTU de comunicação, o Mestre e os escravos devem usar os mesmos parâmetros */

enum {

  COMM_BPS = 9600, /* baud rate */

  MB_SLAVE = 1,          /* endereço do escravo modbus */

  /* cada escravo modbus deve ter um */

  /* endereço único na rede modbus */

  PARITY = 'n',    /* paridade */

  TXEN = 6        /*Definir o pino usado para colocar o driver

   RS485 em modo de transmissão, utilizado

   somente em redes RS485 quando colocar em 0

   ou 1 para redes RS232 */

};
```

```
/* registros do escravo (holding registers)
 *
 * Aqui ficam ordenados todos os registros de leitura e escrita
 * da comunicação entre o mestre e o escravo (SCADA e arduino)
 *
 */

enum {
  MB_PINO_5,      /* Controle do Led no pino  3 (desliga=0 liga=1) offset 0 (p/ data point no scadabr)*/
  MB_PINO_4,      /* Leitura da chave no pino 4 (desliga=0 liga=1) offset 1 (p/ data point no scadabr)*/
  MB_A0,          /* Leitura da entrada analógica 0 (0 a 1023)      offset 2 (p/ data point no scadabr)*/
  MB_REGS         /* número total de registros do escravo */
};

int regs[MB_REGS];
int ledPin5  =  5;
int ledPin13  = 13;
int buttonPin4  = 4;
int buttonState = 0;        // variável para ler e estado do botão

unsigned long wdog = 0;         /* watchdog */
unsigned long tprev = 0;        /* tempo anterior do último comando*/
unsigned long tanalogprev = 0;  /* tempo anterior da leitura dos pinos analógicos*/

void setup()
{
```

```
  /* configura cominicação modbus
   * 9600 bps, 8N1, RS485 network */
  configure_mb_slave(COMM_BPS, PARITY, TXEN);

  pinMode(ledPin5,   OUTPUT);
  pinMode(ledPin13,  OUTPUT);
  pinMode(buttonPin4, INPUT);
}

void loop()
{
  /* verifica se há solicitações do mestre */
  if(update_mb_slave(MB_SLAVE, regs, MB_REGS))
    wdog = millis();

  if ((millis() - tanalogprev) > 1000)  {      /* atualiza as entradas analogica a cada 1 segundo */
    regs[MB_A0] = analogRead(0);          /* ler entrada analógica 0 e salva valor no registro
modbus*/

    tanalogprev = millis();
  }

  buttonState = digitalRead(buttonPin4); // ler estado da chave no pino 4
  regs[MB_PINO_4] = buttonState;               // salva valor no registro modbus
  if (buttonState == HIGH) {            // caso a chave esteja pressionada
    digitalWrite(ledPin13, HIGH);        // liga o Led do pino 13
  }
  else {                                // caso a chave não esteja pressionada
```

```
    digitalWrite(ledPin13, LOW);         // desliga o Led do pino 13
  }

  /* os valores do registro MB_PINO_5 é definido pelo mestre modbus (SCADA) */

  switch(regs[MB_PINO_5]) {
  case 1:
    digitalWrite(ledPin5, HIGH);
    break;
  default: /* apagado */
    digitalWrite(ledPin5, LOW);
  }
}

/**************************************************************************
 * INICIO DAS FUNÇÕES ESCRAVO Modbus RTU
 *************************************************************************/

/* variaveis globais */
unsigned int Txenpin = TXEN;       /*Definir o pino usado para colocar o driver
 RS485 em modo de transmissão, utilizado
 somente em redes RS485 quando colocar em 0
 ou 1 para redes RS232 */


/* Lista de códigos de função modbus suportados. Se você implementar um novo, colocar o seu
código de função aqui! */
enum {
```

```
  FC_READ_REGS   = 0x03,    //Read contiguous block of holding register (Ler um bloco contíguo de registos)

  FC_WRITE_REG  = 0x06,   //Write single holding register (Escrever em um único registro)

  FC_WRITE_REGS = 0x10     //Write block of contiguous registers (Escrever em um bloco contíguo de registos)

};


/* Funções suportadas. Se você implementar um novo, colocar seu código em função nessa matriz! */

const unsigned char fsupported[] = {

  FC_READ_REGS, FC_WRITE_REG, FC_WRITE_REGS };


/* constantes */

enum {

  MAX_READ_REGS = 0x7D,

  MAX_WRITE_REGS = 0x7B,

  MAX_MESSAGE_LENGTH = 256

};



enum {

  RESPONSE_SIZE = 6,

  EXCEPTION_SIZE = 3,

  CHECKSUM_SIZE = 2

};


/* código de exceções */

enum {

  NO_REPLY = -1,
```

```
  EXC_FUNC_CODE = 1,
  EXC_ADDR_RANGE = 2,
  EXC_REGS_QUANT = 3,
  EXC_EXECUTE = 4
};

/* posições dentro da matriz de consulta / resposta */
enum {
  SLAVE = 0,
  FUNC,
  START_H,
  START_L,
  REGS_H,
  REGS_L,
  BYTE_CNT
};



/*
CRC

 INPUTS:
        buf   ->  Matriz contendo a mensagem a ser enviada para o controlador mestre.
        start ->  Início do loop no crc do contador, normalmente 0.
        cnt   ->  Quantidade de bytes na mensagem a ser enviada para o controlador mestre
 OUTPUTS:
        temp  ->  Retorna byte crc para a mensagem.
```

```
COMMENTÁRIOS:

        Esta rotina calcula o byte crc alto e baixo de uma mensagem.

Note que este CRC é usado somente para Modbus, não em Modbus PLUS ou TCP.

*****************************************************************************/


unsigned int crc(unsigned char *buf, unsigned char start,
unsigned char cnt)
{
  unsigned char i, j;
  unsigned temp, temp2, flag;

  temp = 0xFFFF;

  for (i = start; i < cnt; i++) {
    temp = temp ^ buf[i];

    for (j = 1; j <= 8; j++) {
      flag = temp & 0x0001;
      temp = temp >> 1;
      if (flag)
        temp = temp ^ 0xA001;
    }
  }

  /* Inverter a ordem dos bytes. */
  temp2 = temp >> 8;
  temp = (temp << 8) | temp2;
```

```
  temp &= 0xFFFF;

  return (temp);
}

/************************************************************************
 *
 *          As seguintes funções constroem o frame de
 *      um pacote de consulta modbus.
 *
 ************************************************************************/

/*
 * Início do pacote de uma resposta read_holding_register
 */
void build_read_packet(unsigned char slave, unsigned char function,
unsigned char count, unsigned char *packet)
{
  packet[SLAVE] = slave;
  packet[FUNC] = function;
  packet[2] = count * 2;
}

/*
```

```
 * Início do pacote de uma resposta preset_multiple_register
 */
void build_write_packet(unsigned char slave, unsigned char function,
unsigned int start_addr,
unsigned char count,
unsigned char *packet)
{
  packet[SLAVE] = slave;
  packet[FUNC] = function;
  packet[START_H] = start_addr >> 8;
  packet[START_L] = start_addr & 0x00ff;
  packet[REGS_H] = 0x00;
  packet[REGS_L] = count;
}

/*
 * Início do pacote de uma resposta write_single_register
 */
void build_write_single_packet(unsigned char slave, unsigned char function,
unsigned int write_addr, unsigned int reg_val, unsigned char* packet)
{
  packet[SLAVE] = slave;
  packet[FUNC] = function;
  packet[START_H] = write_addr >> 8;
  packet[START_L] = write_addr & 0x00ff;
  packet[REGS_H] = reg_val >> 8;
  packet[REGS_L] = reg_val & 0x00ff;
```

```
}



/*
 * Início do pacote de uma resposta excepção
 */
void build_error_packet(unsigned char slave, unsigned char function,
unsigned char exception, unsigned char *packet)
{
  packet[SLAVE] = slave;
  packet[FUNC] = function + 0x80;
  packet[2] = exception;
}



/*****************************************************************************
 *
 * modbus_query( packet, length)
 *
 * Função para adicionar uma soma de verificação para o fim de um pacote.
 * Por favor, note que a matriz pacote deve ser de pelo menos 2 campos mais do que
 * String_length.
 *****************************************************************************/

void modbus_reply(unsigned char *packet, unsigned char string_length)
{
  int temp_crc;
```

```
  temp_crc = crc(packet, 0, string_length);
  packet[string_length] = temp_crc >> 8;
  string_length++;
  packet[string_length] = temp_crc & 0x00FF;
}



/**************************************************************************
 *
 * send_reply( query_string, query_length )
 *
 * Função para enviar uma resposta a um mestre Modbus.
 * Retorna: o número total de caracteres enviados
 **************************************************************************/

int send_reply(unsigned char *query, unsigned char string_length)
{
  unsigned char i;

  if (Txenpin > 1) { // coloca o MAX485 no modo de transmissão
    UCSR0A=UCSR0A |(1 << TXC0);
    digitalWrite( Txenpin, HIGH);
    delayMicroseconds(3640);  // aguarda silencio de 3.5 caracteres em 9600bps
  }
```

```
  modbus_reply(query, string_length);
  string_length += 2;

  for (i = 0; i < string_length; i++) {
    Serial.write(byte(query[i]));
  }

  if (Txenpin > 1) {// coloca o MAX485 no modo de recepção
    while (!(UCSR0A & (1 << TXC0)));
    digitalWrite( Txenpin, LOW);
  }

  return i;                     /* isso não significa que a gravação foi bem sucedida */
}

/**************************************************************************
 *
 *          receive_request( array_for_data )
 *
 * Função para monitorar um pedido do mestre modbus.
 *
 * Retorna: Número total de caracteres recebidos se OK
 * 0 se não houver nenhum pedido
 * Um código de erro negativo em caso de falha
 **************************************************************************/

int receive_request(unsigned char *received_string)
```

```
{
  int bytes_received = 0;

  /* FIXME: não Serial.available esperar 1.5T ou 3.5T antes de sair do loop? */
  while (Serial.available()) {
    received_string[bytes_received] = Serial.read();
    bytes_received++;
    if (bytes_received >= MAX_MESSAGE_LENGTH)
      return NO_REPLY;  /* erro de porta */
  }

  return (bytes_received);
}


/*****************************************************************************
 *
 *          modbus_request(slave_id, request_data_array)
 *
 * Função que é retornada quando o pedido está correto
 * e a soma de verificação está correto.
 * Retorna: string_length se OK
 * 0 se não
 * Menos de 0 para erros de exceção
 *
 * Nota: Todas as funções usadas para enviar ou receber dados via
 * Modbus devolver esses valores de retorno.
```

```
 *
 ***********************************************************************/

int modbus_request(unsigned char slave, unsigned char *data)
{
  int response_length;
  unsigned int crc_calc = 0;
  unsigned int crc_received = 0;
  unsigned char recv_crc_hi;
  unsigned char recv_crc_lo;

  response_length = receive_request(data);

  if (response_length > 0) {
    crc_calc = crc(data, 0, response_length - 2);
    recv_crc_hi = (unsigned) data[response_length - 2];
    recv_crc_lo = (unsigned) data[response_length - 1];
    crc_received = data[response_length - 2];
    crc_received = (unsigned) crc_received << 8;
    crc_received =
      crc_received | (unsigned) data[response_length - 1];

    /*********** verificar CRC da resposta ************/
    if (crc_calc != crc_received) {
      return NO_REPLY;
    }
```

```
    /* verificar a ID do escravo */
    if (slave != data[SLAVE]) {
      return NO_REPLY;
    }
  }
  return (response_length);
}

/*************************************************************************
 *
 *          validate_request(request_data_array, request_length, available_regs)
 *
 * Função para verificar se o pedido pode ser processado pelo escravo.
 *
 * Retorna: 0 se OK
 * Um código de exceção negativa em caso de erro
 *
 *************************************************************************/

int validate_request(unsigned char *data, unsigned char length,
unsigned int regs_size)
{
  int i, fcnt = 0;
  unsigned int regs_num = 0;
  unsigned int start_addr = 0;
  unsigned char max_regs_num;
```

```
/* verificar o código de função */
for (i = 0; i < sizeof(fsupported); i++) {
  if (fsupported[i] == data[FUNC]) {
    fcnt = 1;
    break;
  }
}
if (0 == fcnt)
  return EXC_FUNC_CODE;

if (FC_WRITE_REG == data[FUNC]) {
  /* Para a função de escrever um reg único, este é o registro alvo.*/
  regs_num = ((int) data[START_H] << 8) + (int) data[START_L];
  if (regs_num >= regs_size)
    return EXC_ADDR_RANGE;
  return 0;
}

/* Para as funções de leitura / escrita de registros, este é o intervalo. */
regs_num = ((int) data[REGS_H] << 8) + (int) data[REGS_L];

/* verifica a quantidade de registros */
if (FC_READ_REGS == data[FUNC])
  max_regs_num = MAX_READ_REGS;
else if (FC_WRITE_REGS == data[FUNC])
  max_regs_num = MAX_WRITE_REGS;
```

```
  if ((regs_num < 1) || (regs_num > max_regs_num))
    return EXC_REGS_QUANT;

  /* verificará a quantidade de registros, endereço inicial é 0 */
  start_addr = ((int) data[START_H] << 8) + (int) data[START_L];
  if ((start_addr + regs_num) > regs_size)
    return EXC_ADDR_RANGE;

  return 0;                    /* OK, sem exceção */
}



/*****************************************************************************
 *
 *          write_regs(first_register, data_array, registers_array)
 *
 *          escreve nos registradores do escravo os dados em consulta,
 * A partir de start_addr.
 *
 * Retorna: o número de registros escritos
 *****************************************************************************/

int write_regs(unsigned int start_addr, unsigned char *query, int *regs)
{
  int temp;
  unsigned int i;
```

```
  for (i = 0; i < query[REGS_L]; i++) {
    /* mudar reg hi_byte para temp */
    temp = (int) query[(BYTE_CNT + 1) + i * 2] << 8;
    /* OR com lo_byte           */
    temp = temp | (int) query[(BYTE_CNT + 2) + i * 2];

    regs[start_addr + i] = temp;
  }
  return i;
}

/**************************************************************************
 *
 *          preset_multiple_registers(slave_id, first_register, number_of_registers,
 * data_array, registers_array)
 *
 *          Escreva os dados na matriz dos registos do escravo.
 *
 **************************************************************************/

int preset_multiple_registers(unsigned char slave,
unsigned int start_addr,
unsigned char count,
unsigned char *query,
int *regs)
{
```

```
  unsigned char function = FC_WRITE_REGS;   /* Escrever em múltiplos registros */
  int status = 0;
  unsigned char packet[RESPONSE_SIZE + CHECKSUM_SIZE];

  build_write_packet(slave, function, start_addr, count, packet);

  if (write_regs(start_addr, query, regs)) {
    status = send_reply(packet, RESPONSE_SIZE);
  }

  return (status);
}

/**************************************************************************
 *
 * write_single_register(slave_id, write_addr, data_array, registers_array)
 *
 * Escrever um único valor inteiro em um único registo do escravo.
 *
 **************************************************************************/

int write_single_register(unsigned char slave,
unsigned int write_addr, unsigned char *query, int *regs)
{
  unsigned char function = FC_WRITE_REG; /* Função: Write Single Register */
  int status = 0;
```

```
  unsigned int reg_val;
  unsigned char packet[RESPONSE_SIZE + CHECKSUM_SIZE];

  reg_val = query[REGS_H] << 8 | query[REGS_L];
  build_write_single_packet(slave, function, write_addr, reg_val, packet);
  regs[write_addr] = (int) reg_val;
  /*
       written.start_addr=write_addr;
   written.num_regs=1;
   */
  status = send_reply(packet, RESPONSE_SIZE);

  return (status);
}


/*****************************************************************************
 *
 *          read_holding_registers(slave_id, first_register, number_of_registers,
 * registers_array)
 *
 * lê os registros do escravo e envia para o mestre Modbus
 *
 *****************************************************************************/

int read_holding_registers(unsigned char slave, unsigned int start_addr,
```

```
unsigned char reg_count, int *regs)
{
  unsigned char function = 0x03;          /* Função 03: Read Holding Registers */
  int packet_size = 3;
  int status;
  unsigned int i;
  unsigned char packet[MAX_MESSAGE_LENGTH];

  build_read_packet(slave, function, reg_count, packet);

  for (i = start_addr; i < (start_addr + (unsigned int) reg_count);
                  i++) {
     packet[packet_size] = regs[i] >> 8;
    packet_size++;
    packet[packet_size] = regs[i] & 0x00FF;
    packet_size++;
  }

  status = send_reply(packet, packet_size);

  return (status);
}


void configure_mb_slave(long baud, char parity, char txenpin)
{
  Serial.begin(baud);
```

```
  switch (parity) {
  case 'e': // 8E1
    UCSR0C |= ((1<<UPM01) | (1<<UCSZ01) | (1<<UCSZ00));
    //      UCSR0C &= ~((1<<UPM00) | (1<<UCSZ02) | (1<<USBS0));
    break;
  case 'o': // 8O1
    UCSR0C |= ((1<<UPM01) | (1<<UPM00) | (1<<UCSZ01) | (1<<UCSZ00));
    //      UCSR0C &= ~((1<<UCSZ02) | (1<<USBS0));
    break;
  case 'n': // 8N1
    UCSR0C |= ((1<<UCSZ01) | (1<<UCSZ00));
    //      UCSR0C &= ~((1<<UPM01) | (1<<UPM00) | (1<<UCSZ02) | (1<<USBS0));
    break;
  default:
    break;
  }

  if (txenpin > 1) { // pino 0 & pino 1 são reservados para RX/TX
    Txenpin = txenpin; /* definir variável global */
    pinMode(Txenpin, OUTPUT);
    digitalWrite(Txenpin, LOW);
  }

  return;
}
```

```
/*
 * update_mb_slave(slave_id, holding_regs_array, number_of_regs)
 *
 * verifica se há qualquer pedido válido do mestre modbus. Se houver,
 * executa a ação solicitada
 */

unsigned long Nowdt = 0;
unsigned int lastBytesReceived;
const unsigned long T35 = 5;

int update_mb_slave(unsigned char slave, int *regs,
unsigned int regs_size)
{
  unsigned char query[MAX_MESSAGE_LENGTH];
  unsigned char errpacket[EXCEPTION_SIZE + CHECKSUM_SIZE];
  unsigned int start_addr;
  int exception;
  int length = Serial.available();
  unsigned long now = millis();

  if (length == 0) {
    lastBytesReceived = 0;
    return 0;
  }

  if (lastBytesReceived != length) {
```

```
  lastBytesReceived = length;
  Nowdt = now + T35;
  return 0;
}
if (now < Nowdt)
  return 0;

lastBytesReceived = 0;

length = modbus_request(slave, query);
if (length < 1)
  return length;


exception = validate_request(query, length, regs_size);
if (exception) {
  build_error_packet(slave, query[FUNC], exception,
  errpacket);
  send_reply(errpacket, EXCEPTION_SIZE);
  return (exception);
}


start_addr = ((int) query[START_H] << 8) +
  (int) query[START_L];
switch (query[FUNC]) {
case FC_READ_REGS:
```

```
    return read_holding_registers(slave,
    start_addr,
    query[REGS_L],
    regs);
    break;
  case FC_WRITE_REGS:
    return preset_multiple_registers(slave,
    start_addr,
    query[REGS_L],
    query,
    regs);
    break;
  case FC_WRITE_REG:
    write_single_register(slave,
    start_addr,
    query,
    regs);
    break;
  }
}
```

Referências

Arduino:
http://www.arduino.cc/

SCADABR :
http://www.scadabr.com.br/

JPMZOMETA:
https://sites.google.com/site/jpmzometa/arduino-mbrt/arduino-modbus-slave

Fórum SCADABR no Laboratório de Garagem:
http://labdegaragem.com /forum/topics/scadabr

MODBUS:
http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf

RS485:
http://www.lammertbies.nl/comm/info/RS-485.html
http://www.maxim-ic.com/app-notes/index.mvp/id/763

Videos no Youtube:
http://www.youtube.com/watch?v=aXcgloY4a80
http://www.youtube.com/watch?v=hHhWS2E2BmQ&feature=player_embedded
http://www.youtube.com/watch?v=nHSG7yrs4JE&feature=channel&list=UL
http://www.youtube.com/watch?v=N4vE0F8A1II&feature=relmfu
http://www.youtube.com/watch?v=j-OW5jYRWYk&feature=relmfu